

phy++ v1.0

Reference manual

Corentin Schreiber |
May 22, 2015

Contents

1	Introduction	4
1.1	Description of the library	4
1.1.1	Overview	4
1.1.2	Why write something new?	5
1.1.3	Why C++?	7
1.2	Supported systems	8
1.3	Installing	9
1.4	Quick-start guide	9
1.4.1	For users new to C++	9
1.4.2	For users familiar with C++	9
2	Core library	10
2.1	Overview	10
2.1.1	Operator overloading	11
2.1.2	Multi-dimensional indexing	12
2.1.3	Vector views	13
2.2	The vector class	15
2.2.1	Member variables	15
2.2.2	Constructors and assignment	16
2.2.3	Member functions	20
2.2.4	Indexing	25
2.2.5	Operators	28
2.3	The view class	29
2.3.1	Member functions	29
2.4	Metaprogramming traits and functions (WIP)	29
3	Support libraries	30
3.1	Generic vector functions	34

3.1.1	Range-based iteration	34
3.1.2	Index manipulation	35
3.1.3	Integer sequences	36
3.1.4	Rearranging elements	37
3.1.5	Finding elements	38
3.1.6	Modifying dimensions	43
3.1.7	Adding/removing elements	45
3.1.8	Error checking	46
3.2	Parsing command line arguments	47
3.3	String manipulation	51
3.3.1	String conversions	51
3.3.2	String operations	54
3.4	OS interaction	66
3.5	File input/output	66
3.5.1	File system	66
3.5.2	ASCII table input/output	71
3.6	Printing to the terminal	77
3.7	Measuring time	77
3.8	Mathematics	77
3.8.1	Low level mathematics	79
3.8.2	Sequences and bins	79
3.8.3	Randomization	80
3.8.4	Reduction	80
3.8.5	Interpolation	81
3.8.6	Calculus	82
3.8.7	Algebra	82
3.8.8	Fitting	83
3.8.9	Geometry	83
3.8.10	Debug functions	84
3.9	Parallel execution	84
3.10	FITS input/output	84
3.10.1	Generic header functions	85
3.10.2	FITS images input/output	85
3.10.3	WCS coordinates	86
3.10.4	FITS tables input/output	86
3.11	Image processing	86
3.12	Astrophysics	87
3.12.1	PSF fitting	87

3.12.2	Cosmology	87
3.12.3	Fluxes, magnitudes and luminosities	88
3.12.4	Sky coordinates	88
3.12.5	Catalog management	89
3.12.6	Image stacking	89
3.12.7	Template fitting	90
4	Tools	91
4.1	Astrophysics	92
4.1.1	angcorrel	92
4.1.2	catinfo	92
4.1.3	deg2sex and sex2deg	92
4.1.4	findsrc	92
4.1.5	fluxcube	92
4.1.6	getgal	92
4.1.7	photinfo	92
4.1.8	psffit	92
4.1.9	qstack2	92
4.1.10	qxmacth2	92
4.1.11	randsrc	92
4.1.12	subsrc	92
4.2	FITS and ASCII	92
4.2.1	fits2ascii	92
4.2.2	fitstool	92
4.2.3	imgtool	92
4.2.4	qconvol	92
4.2.5	remcol	92
	Index	93

Chapter 1

Introduction

1.1 Description of the library

1.1.1 Overview

phy₊₊ is a set of library and tools built to provide user-friendly vector data manipulation, as offered in interpreted languages like IDL¹, its open source clone GDL², or python & numpy³, but with the added benefit of C++: increased robustness, and optimal speed.

The library can be split into two components: the *core* library (2) and the *support* library (3). The core library introduces the *vector* type, which is at the heart of *phy₊₊*, while the support library provides functions and other tools to manipulate these vectors and do some common tasks. You can think of the core library as “the language”, and the support library as “the function library”.

Below is an code sample written in *phy₊₊* that illustrates the most basic functionalities.

```
vec2f img = fits::read("img.fits"); // read a FITS image
img -= median(img);                // subtract the median of the whole image
float imax = max(img);              // find the maximum of the image
vec1u ids = where(img > 0.5*imax); // find pixels at least half as bright
float sum = total(img[ids]);         // compute the sum of these pixels
img[ids] = log(img[ids]/sum);        // modify these pixels with a logarithm
fits::write("new.fits", img);       // save the modified image to a FITS file
```

¹<http://www.exelisvis.com/ProductsServices/IDL.aspx>

²<http://gnudatalanguage.sourceforge.net/>

³<http://www.numpy.org/>

1.1.2 Why write something new?

The immediate goal of *phy++* is to provide a syntax as close as possible to that of IDL. IDL is an interpreted language that is widely used in the scientific community, in particular in astrophysics. Born in the late 1970s, this language provides intuitive manipulation of large arrays of data using vectorized operations: applying an operation on a given array does not require the user to write a loop to iterate over its elements and apply the operation. This leads to very concise code that is easy to write and read. Unfortunately, IDL suffers from a number of problems. I will start with the *political* and *ethical* problems.

- It is a proprietary, mostly⁴ closed-source program. This means that IDL is a black box and that people using it have no choice but to rely on the IDL developers for writing accurate code. While there is an extensive documentation, the algorithms used by the procedures are not always described. This is hardly acceptable for scientific code.
- IDL, like C++, combines several languages into one: a functional language and an object-oriented language. It also contains a huge support library providing many features (having used IDL for more than two years, I could not list them all). For this reason, and because it is proprietary, maintaining this language and adding new features costs a lot of money to its owner, Exelis. This money, in turn, is provided by science labs all around the world, who pay a yearly fee for a bunch of IDL licenses. This is totally fine in itself, but the fact is that most IDL users I have seen only make use of a small sub-set of IDL, one that has barely evolved in twenty years. In this context, the price that is paid is not justified.
- On top of that, the licensing model is that of *floating* licenses: only a fixed, maximum number of simultaneously running IDL instances is allowed in the whole lab. With the now common budget restrictions in research, labs typically buy fewer licenses than there are users. Even worse, it is often needed to run multiple instances of IDL on a single computer, e.g., when working on two projects simultaneously. This will consume two licenses, even though there is a single user. This leads to silly situations, typically when approaching specific deadlines (e.g., deadlines for requesting observing time on large telescopes) where everyone needs to use IDL at the same time, but there is not enough license available. Even worse, we have seen cases in our lab of users being unable to run IDL on their new shiny computer because of incompatibility, not with IDL itself, but with the licensing software. Lastly,

⁴The procedures from the IDL library that are written in IDL language are actually open-source, but all the procedures written in native language are compiled and only the binary is provided.

it should be noted that this licensing model relies on having network connection with a license server. This means that one cannot use IDL while traveling unless a proper SSH tunneling is in place.

These issues can be solved by switching to one of the free and open-source alternatives, like GDL. The downside is that these implementations are lacking behind IDL in terms of features, as some useful functions are still to be implemented. Worse, some functions cannot *legally* be implemented because they would violate IDL's copyright.

But that's only half of the story. Indeed, IDL and GDL also suffer from technical issues. I will list below the most important ones.

- Designed in the 1970s, IDL was born in an era where the available RAM was scarce, and that great care had to be taken to consume as few bytes of memory as possible. For this reason, the default integer type in IDL is a **short**, i.e., it occupies only two bytes in memory, while most languages (including some that are older than IDL itself) encode their integers on four bytes by default. The biggest issue with this choice is that the largest number one can store in a **short** is 32768. Being the default integer type, this creates quite a few surprises to the unexperienced user, and will fool even the expert from time to time.
- IDL is an interpreted language, meaning that the code you write is continuously read and interpreted by the IDL executable. While this is not an issue if you make good use of vectorization (the art of writing IDL code), performances are severely degraded once you write loops explicitly, because the content of the loop has to be *interpreted* and then *executed* on each iteration. And this is sometimes unavoidable.
- Like many interpreted languages, IDL is dynamically typed. This means that the type of a variable can change from one line to another, and that a variable containing a string can be assigned a number. While sometimes convenient, this comes at a cost: performance. And the fact is that in most IDL programs this feature is not used.
- But worse than dynamic typing, and this is my main concern, variables in IDL are not *declared* before they are used. This means that if you do a typo in the name of one of your variables, chances are that the code will still run. Indeed, IDL cannot know that this was not intended, and will think that you want to create a new variable. It will then do its best to carry on, and the result will be unpredictable. This, together with the fact that variables are almost not *scoped* (i.e., a variable created inside a **for** loop is still valid outside of the loop) makes it very easy to write confusing and buggy code. The most frightening part is that, in a good fraction of the cases, the output will be meaningful, and you can go on with your calculation never realizing that something went wrong. And publish that.

Switching to more modern interpreted languages like python or Julia⁵ would solve a few of these issues, in particular the first one. But the other items on this list are unfortunately inherent to most interpreted languages⁶. To avoid these traps, the only solution today is to use *statically typed*, compiled languages, like C++.

Now, there are already some libraries in C++ that are addressing the topic of vector data manipulation. One can cite **Eigen** or the more recent **blaze-lib**. These are wonderful libraries that have inspired *phy++* in some way, but their issue is that they are more oriented toward algebra, meaning that they have vectors and matrices, but no data type for arrays of higher dimensions (i.e., tensors⁷).

Therefore, seeing that a gap had to be filled, *phy++* was created.

1.1.3 Why C++?

There are many different compiled languages that offer similar or better performances than C++. In particular, the most famous ones are Fortran and C. C is impractical to use because it has not been developed with user-friendliness in mind, and no mechanism exist to improve that. This is a system language, and it does that perfectly, but not much more. Fortran is known as the fastest of all, and it is particularly well suited for numerical analysis. While few languages as harder to read than Fortran 77, things have become much better since Fortran 90 (which is not used as often as it should be). However, Fortran is relatively bad at doing anything else than numerical analysis, which is annoying the moment you want to do something that is a bit off the tracks. C++ on the other hand, with all its disadvantages, is probably the best fit thanks to its almost unlimited capacity for adaptation. And it also happens to be the language I am most familiar with.

Since the beginning, C++ has always been good at performances, first because it is a language that compiles directly into assembly, but also thanks to its philosophy: “you only pay for what you ask for”. But its main disadvantage is its *complexity*: it contains almost the whole C language, plus all the layers that were added on top of it, one year after another, starting from classes, exceptions, then templates. The end result is that it is a challenging task to master all the aspects of this language.

But the good news is: you do not have to master all of C++, and for your sanity you probably should not. Indeed, there is a number of *sub-languages* made out of a subset C++ that are completely self-sufficient, i.e. you can use them to write any program. In other words, there are multiple, very different ways of writing the same program in C++. Typically, modern programs only use a small fraction of the whole language, e.g., leaving

⁵<http://julialang.org/>

⁶The best counter example is probably Java.

⁷Eigen actually has a tensor module, but it is unsupported.

aside most of what was inherited from C (arrays, raw pointers, explicit memory management, etc.). A special class of such sub-languages are those that are tailored specifically to address a given task, as opposed to being open to any purpose. These are called *domain-specific languages* (DSL), and only require learning a few of C++’s rules and concepts, plus the rules introduced by the sub-language itself. The *phy₊₊* library is an example of such domain-specific languages, its domain being vector data manipulation.

In short, although C++ is a very complex language, it is only necessary to learn a fraction of it to be able to use *phy₊₊* correctly. For example, you will find in 1.4.1 a self-consistent tutorial to get started with the library that does not require any preliminary knowledge of C++. Of course, the more you will know about C++, the more you will be able to take advantage of all the features of *phy₊₊* in an optimal way.

1.2 Supported systems

The combination of operating systems and compilers on which this library has been tested is reported in the following table. If you have issues and your system is reported in this table as working, then there is probably something wrong in your configuration (e.g., a missing library). If you are using a system combination that is not listed here: if you managed to get it to work, please drop me an email and I will add it to the list; else, if you failed to compile or run programs with the library, please open a new issue on [github](https://github.com/cschreib/phypp/issues)⁸ and we will try to get it fixed together.

Operating system	Compiler	Status	Notes
Linux Mint 15	gcc-4.8	works	^a
—	gcc-4.7	works	^a
—	clang-3.3	works	^a
Linux Mint 17	gcc-4.8	works	^b
—	clang-3.3	works	^b
—	clang-3.4	works	^b
—	clang-3.5	works	^b
Ubuntu 14.04	gcc-4.8	works	^c
Debian 4.6.3	gcc-4.7	works	^c
OSX 10.9.5	xcode-6.0	works	^c

^alibclang in the repository is too old. Use a more recent version from [ppa:h-rayflood/llvm](https://github.com/llvm/llvm-project).

^blibclang in the repository crashes on valid C++ code. Fixed in libclang-3.5.

^cNot all dependencies were tested.

⁸<https://github.com/cschreib/phypp/issues>

1.3 Installing

1.4 Quick-start guide

1.4.1 For users new to C++

1.4.2 For users familiar with C++

Chapter 2

Core library

2.1 Overview

At the core of the *phy++* library is the *vector* class. This is basically an enhanced `std::vector`¹, and it therefore shares most of its features and strengths. In particular, a vector can contain zero, one, or as many elements as your computer can handle. Its size is defined at *run-time*, meaning that its content can vary depending on user input, but also that a vector can change its total number of elements at any time. These elements are stored contiguously in memory, which provides optimal performances in most situations. Lastly, a vector is an homogeneous container, meaning that a given vector can only contain a single type of elements (e.g., **int** or **float**, but not both at the same time).

Like most advanced C++ libraries, *phy++* is essentially *template* based. This means that most of the code is written to work for *any* type T, e.g., **int**, **float**, `std::string`, or whatever you need. However, while templates are a fantastic tool for library writers, they can easily become a burden for the *user* of the library. The good thing is, since *phy++* is a numerical analysis library, we know in advance what types will most often be stored inside the vectors. So for this reason, to reduce typing and enhance readability, we introduce type aliases for the most commonly used vector types:

- `vec1f`: vector of **float**,
- `vec1d`: vector of **double**,
- `vec1cf`: vector of `std::complex<float>`,
- `vec1cd`: vector of `std::complex<double>`,

¹In fact, `std::vector` is used to implement the *phy++* vectors internally.

- `vec1i`: vector of `int` (precisely, `int_t = std::ptrdiff_t`),
- `vec1u`: vector of `unsigned int` (precisely, `uint_t = std::size_t`),
- `vec1b`: vector of `bool`,
- `vec1s`: vector of `std::string`,
- `vec1c`: vector of `char`.

On top of the `std::vector` interface, the *phy++* vector adds some extra functionalities. The most important ones are operator overloading, multi-dimensional indexing, and vector views.

2.1.1 Operator overloading

The only thing you can do to operate on all the elements of an `std::vector` is to iterate over these elements explicitly, either using a C++11 range-based loop, or using indices:

```
// Goal: multiply all elements by two.
std::vector<float> v = {1,2,3,4};

// Either using a range-based loop,
for (float& x : v) {
    x *= 2;
}

// ... or an index-based loop.
for (std::size_t i = 0; i < v.size(); ++i) {
    v[i] *= 2;
}
```

While this is fairly readable (especially the first version), it is still not very concise and expressive. In *phy++*, we have *overloaded* the usual mathematical operators on our vector type, meaning that it is possible to write the above code in a much simpler way:

```
// Using phy++ vector.
vec1f w = {1,2,3,4};
w *= 2; // {2,4,6,8}
```

Not only this, but we can perform operations on a pair of vectors in the same way:

```
// Goal: sum the content of the two vectors.
vec1f x = {1,2,3,4}, y = {4,3,2,1};
vec1f z = x + y; // {5,5,5,5}
```

Warning

The only issue with operator overloading concerns the hat operator `^`. In most languages, this operator is used for exponentiation, e.g., `4^2 == 16`. However, in C++ this value is actually 6, because the hat operator is the binary XOR (exclusive-or). Even worse, the hat operator in C++ does not have the same *precedence* as in regular mathematics: it has a lower priority than any other mathematical operator. Both these reasons make it unwise to overload the hat operator in *phy++*. In order to perform exponentiation, you will have to use a dedicated function such as `pow(4,2)` (3.8).

2.1.2 Multi-dimensional indexing

The standard `std::vector` is a purely linear container: one can access its elements using `v[i]`, with `i` ranging from 0 up to `std::vector::size()-1` (included). However, the *phy++* vector allows N-dimensional indexing, i.e., using a group of indices to identify one element. For example, this is particularly useful to work on images, which are essentially 2-dimensional objects where one identifies a given pixel by its coordinates `x` and `y`. The natural syntax for this indexing would be to write `img[x,y]`. This syntax is valid C++ code, but unfortunately will not do what you expect² and there is no sane way around it. The alternative we chose here is to write instead `img(x,y)`. While it is not as semantically clear as using brackets, it has the nice advantage of being compatible with the IDL syntax.

The multi-dimensional nature of a vector is determined at *compile time*, i.e., it cannot be changed after the vector is declared. By default, a vector is mono-dimensional. To use the above feature, one needs to specify the number of needed dimensions in the type of the vector. For example, a 2D image of `float` will be declared as `vec2f`. These type aliases are provided for dimensions up to 6. Here is an example of manipulation of a 2D matrix:

```
// Create a simple matrix.
vec2f m = {{1,2,3}, {4,5,6}, {7,8,9}};
```

```
// Index ordering is similar to C arrays: the last index
```

²This will call the *comma* operator, which evaluates both elements `x` and `y` and returns the last one, i.e., `y`. So this code actually means `img[y]`. With proper configuration, most compiler will warn about this though, since in this context `x` is a useless statement, so you should be safe should you make this mistake.

```

// is contiguous in memory. Note that this is *opposite*
// to the IDL convention.
m(0,0); // 1
m(0,1); // 2
m(1,0); // 4

// It is still possible to access elements as if in a
// "flat" vector
m[0]; // 1
m[1]; // 2
m[3]; // 4

```

Advanced

If for some reason you need to use more than 6 dimensions, or if you need to declare a vector of some type which is not covered above, you can always fall back to the full template syntax:

```

// Need 12 dimensions?
using vec12f = vec<12, float>;
// Need 512 bit of floating point precision?
using vec3f512 = vec<3, mpfr::real<512>>;

```

The hard limit on the number of dimensions will then depend on your compiler (each dimension involves an additional level of template recursion). The C++ standard does not guarantee anything, but you should be able to go as high as 256 on all major compilers. Beyond this, you should probably see a doctor first.

As for the types allowed inside the vector, there is no explicit restriction. However, some features may obviously not be available depending on the capabilities of your type (e.g., if your type has no **operator***, you will not be able to multiply together vectors of this type). Lastly, the *phy++* vector shares the same restrictions as the `std::vector` regarding the *copyable* and *movable* capabilities of the stored type.

2.1.3 Vector views

We have seen that, instead of accessing each element individually, we can use operator overloading to perform simple operations on all the elements of the vector at once: `w *= 2`. We can also, like with `std::vector`, modify each element individually, knowing their indices: `w[2] *= 2`. One last important feature allowed by the *phy++* vector is that one can create a *view* inside a vector. Each element of the view is actually a *reference* to an

element in the original vector, and modifying the elements of the view actually modifies the elements in the original vector. As you will see later, views actually share most of the interface and capabilities of true vectors, so that most generic codes that work with vectors will also work with views.

```
// Create a simple vector.
vec1f w = {1,2,3,4,5,6};

// We want to "view" only the second, the third and the
// fifth elements. So we first create a vector containing
// the corresponding indices.
vec1u id = {1,2,4};

// Then we create the view.
// Note the usage of "auto" there. The type of the view is
// complex, and it is better not to worry about it.
auto v = w[id];

// Now we can modify these elements very simply, as if they
// were part of a real vector.
v;           // {2,3,5}
v *= 2;      // {4,6,10}
w;           // {1,4,6,4,10,6}
w[1] = 99;   // {1,99,6,4,10,6}
v;           // {99,6,10}
```

Warning

It is important to note that, since a view keeps references to the elements of the original vector, the lifetime of the view must not exceed that of the original vector. Else, it will contain *dangling references*, i.e. pointers to unused memory, and this should be avoided at all cost. In fact, views are not meant to be stored into named variables like in the above example. Most of the time, one will use them as temporary variables, e.g.:

```
vec1f w = {1,2,3,4,5,6};
vec1u id = {1,2,4};
w[id] *= 2; // {1,4,6,4,10,6}
```

2.2 The vector class

The full type of the vector class is

```
template<std::size_t Dimension, typename ElementType>
struct vec<Dimension, ElementType>;
```

In the rest of this document, `Dimension` will usually just be called `D` and `ElementType` will be shortened to `T`.

2.2.1 Member variables

A vector only contains two member variables:

- `std::vector<T> data`

This is the underlying `std::vector` containing the elements of the vector. It is exposed to the public interface for simplicity, but on most occasions it should *not* be used directly. In fact, it may become part of the private interface in the future, so you should not rely on its existence.

Advanced

Concerning `bool` vectors. We do not use `std::vector<bool>`, since it is a very special case of `std::vector`: the C++ standard specifies that the `bool` specialization does not store `bool` elements, but is actually implemented like a *bit field*. This is essentially to save memory: a `bool` in C++ occupies 8 bits of memory, like a `char`, even though it can only carry a single bit of information. This is due to *memory alignment* issues, which are inherent to the CPU architecture (the address of individual values in memory are supposed to be multiples of 8 bits, or one byte). In a bit field however, 8 `bool`s are stored in a single `char`, and bitwise operators are used to read and write individual `bool`s. While more memory efficient, it is also slower, and involves a whole machinery to trick the user into thinking that none of this is happening. For this reason, `bool` vectors are implemented with `std::vector<char>` in *phy++*, with one `char` containing one `bool`. This is completely transparent to the library user though, since `char&` is casted into `bool&`, and vice versa, at the boundary of the vector interface.

- `std::array<std::size_t,D> dims`

This variable contains the dimensions of this vector (useful only for multidimensional vectors). On no occasion should you modify this variable yourself: you

should only read its content³. To change the dimensions of a vector, either use `resize(...)` (2.2.3) or assign it another vector (2.2.2).

Example

```
// Create a 2D image with 256 x 128 pixels
vec2f img(256,128);
img.dims[0]; // 256
img.dims[1]; // 128
for (std::size_t x = 0; x < img.dims[0]; ++x)
for (std::size_t y = 0; y < img.dims[1]; ++y) {
    img(x,y) = 3.1415;
}
```

2.2.2 Constructors and assignment

There are various ways to construct a new vector, or assign it some value. In the following, we will cover the various *constructors* and the associated *assignments* when applicable.

- `vec::vec();` // *The default constructor*

Like `std::vector`, the `phy++` vector is *default constructible*. By default it is in a valid state where the vector does not contain any element.

Example

```
vec1f w; // default constructor: w is empty
```

- **explicit** `vec::vec(...);` // *The dimension constructor*

The `std::vector` has a constructor to create a new vector of a given size. The `phy++` vector also offers this feature, however it is rendered a bit more complex by the possibility of having multi-dimensional vectors. In particular, it is possible to specify the size either by giving all the dimensions one by one, or by specifying

³This statement is actually a bit bold, but is mostly true. The only reason why this variable is made public is for optimization purposes. On occasions, it can be noticeably faster to manage manually the growth of a vector, and update the dimensions afterwards. This is often done within the core library, but should rarely be done otherwise.

some (or all) as a `std::array`. In all cases, the vector is populated with the number of requested elements, and all these elements are *default initialized* (i.e., integers and floats are initialized to 0, booleans to **false**, and `std::string` are empty). This constructor is declared **explicit** to prevent interference with the other constructors and assignments.

Example

```
vec1f w(10);      // w contains 10 objects all equal to 0
vec2d z(5,4);     // z contains 5*4=20 objects all equal to 0
vec2s x(z.dims);  // x has the same dimensions as z
                  // but contains strings, all empty
vec3b y(z.dims, 8); // y has the same dimensions as z,
                  // plus an extra dimension of length 8
```

- `vec::vec(nested_initializer_list<D,T>);` // The list constructor

Like `std::vector`, the *phy++* vector can be initialized from a list of values. This usually requires the usage of `std::initializer_list<T>`, however here we also have to support multi-dimensional vectors, hence we need initializer lists of initializer lists of ..., and `nested_initializer_list<D,T>` is just that. Note that, since C++ is a row-major language, the most nested lists correspond to the last index in a multi-dimensional vector.

Example

```
vec1f w({1,2,3}); // 1D list constructor: {1,2,3}
vec2f z({{1,2,3},{6,5,4}}); // 2D list constructor: {{1,2,3},{6,5,4}}
z(0,0); // 1
z(0,1); // 2
z(0,2); // 3

// Assignment
w = {4,5,6};
z = {{1,2}, {3,4}, {5,6}}; // dimensions can change through assignment
z(0,0); // 1
z(1,0); // 3
z(2,0); // 5
```

- `vec::vec(const vec&); // The copy constructor`

Like `std::vector`, the `phy++` vector is *copiable*, meaning that one can duplicate the content of an existing vector inside another vector by copy. Note that this constructor is only valid for copying vectors of the *same* type. If the type is different, then another constructor is called (conversion constructor, see below).

Example

```
vec1f w = {1,2,3};
vec1f z(w); // copy constructor: {1,2,3}

// Assignment
z = w;
```

- `vec::vec(vec&&); // The move constructor`

Like `std::vector`, the `phy++` vector is *movable*, meaning that one can move the content of an existing vector that is going to be destroyed inside another vector. This is an optimized copy for temporary variables (C++11 move semantics). You need not explicitly ask for either the copy or the move constructor, as they will automatically be chosen by the compiler. For you, this is transparent (but the performance boost is large).

Example

```
vec1f z(vec1f{1,2,3}); // move constructor: {1,2,3}
// Here, a temporary vector is created with vec1f{1,2,3}.
// This temporary vector is then *moved* inside z.

// Assignment
z = vec1f{4,5,6};
```

- `vec::vec(const vec<D,OtherT>&); // The conversion constructor`

C++ supports implicit conversion between all the built-in types. In particular, it is possible to write:

```
int i = 0;
float f = i; // int to float
```

```
bool b1 = i; // int to bool, mostly for interoperability with C
bool b2 = f; // float to bool, not sure if that makes sense...
```

While this is very convenient in most cases, it can also lead to dangerous silent conversions, such as the **float** to **bool** conversion. This is, somehow, a legacy of C. In *phy++* we decided to also support such implicit conversions. They make the code much easier to read, but the price to pay is that sometimes we do some conversions which are not necessary, and we do not realize it because they are implicit. However, we decided to disable implicit conversion to and from **bool** vectors, since it could lead to subtle bugs that are difficult to trace. It is still possible to do the conversion to **bool** using explicit

Example

```
vec1i w = {1,2,0};
vec1f z(w); // int to float {1,2,0}
vec1b y(w); // error: cannot convert int to bool
vec1s x(w); // error: cannot convert int to std::string

vec1b b = {true,true,false};
vec1i s(b); // bool to int {1,1,0}

// Assignment
z = w;
y = w;          // error: cannot convert int to bool
x = w;          // error: cannot convert int to std::string

s = b;          // error: implicit conversion is not allowed for bool
s = vec1i(b); // ok: bool to int {1,1,0}
```

- `vec::vec(const vec<D,T*>&);` // The view constructor

The last constructor on the list is the view constructor. It allows implicit conversion of a view (2.3) into a new, independent vector.

Example

```
vec1i w = {1,2,3};
```

```
vec1u id = {0,1};  
vec1i z(w[id]); // {1,2}  
  
// Assignment  
z = w[id];
```

2.2.3 Member functions

- **bool** `vec::empty()` **const**

This function will return **true** if this vector contains at least one element, and **false** otherwise. In particular, **true** will be returned for default constructed vectors, and after a call to `vec::clear()`.

Example

```
vec1i v;  
v.empty(); // true  
vec1i w = {1,2,3};  
w.empty(); // false
```

- `std::size_t` `vec::size()` **const**

This function will return the total number of elements in this vector. If the vector is empty, then the function returns 0. If the vector is multidimensional, then the function returns the product of all the dimensions.

Example

```
vec1i v;  
v.size(); // 0  
vec1i w = {1,2,3};  
w.size(); // 3  
vec2i z = {{1,2,3}, {4,5,6}};  
z.size(); // 6
```

- **void** `vec::resize(...)`

This function can be used to explicitly change the size of a vector. The parameters it accepts are the same as the dimension constructor (2.2.2), i.e., either integral values

for individual dimensions, or an `std::array` containing multiple dimensions, or any combination of these. However, the total number of dimensions of the vector must remain unchanged.

After the vector has been resized, its content will have changed. For a monodimensional vector, if the resize operation *decreased* the total number of elements, then the last elements will be erased, but the other ones will remain untouched. On the other hand, if the resize operation *increased* the total number of elements, all the previous elements are unchanged, and new elements are inserted at the end of the vector, default constructed (i.e., zeroes for integral types, etc.). For a multidimensional vector, its content is left in an *undefined state*, and can be assumed to be destroyed⁴.

Example

```
vec1i v = {1,2,3};
v.resize(5); // {1,2,3,0,0}
v.resize(2); // {1,2}
v.resize(3); // {1,2,0}

vec2f w = {{1,2},{2,3}};
w.resize(2,3); // w has been resized, but its content is unspecified
w(0,0); // ?
```

- **void** `vec::clear()`

This function removes all elements from the vector, and sets all dimensions to zero.

Example

```
vec1i v = {1,2,3};
v.clear(); // v is now empty
v.empty(); // true
v.size(); // 0
v.dims; // {0}
```

- **T&** `vec::back()`

⁴This is just out of laziness. In the future, this will probably be specified to behave like the monodimensional vector. The issue is that it involves non-trivial reshuffling of the elements to preserve the original structure after the dimensions have changed. I have not needed this feature so far.

This function is only available for monodimensional vectors. It returns the last element of the vector. Will crash if called on an empty vector.

Example

```
vec1i v = {1,2,3};  
v.back(); // 3  
v.back() == v[v.size()-1]; // always true
```

- `T& vec::front()`

This function is only available for monodimensional vectors. It returns the first element of the vector. Will crash if called on an empty vector.

Example

```
vec1i v = {1,2,3};  
v.front(); // 1  
v.front() == v[0]; // always true
```

- **void** `vec::push_back(...)`

The behavior of this function is different for monodimensional and multidimensional vectors. For monodimensional vectors, this function appends a new element at the end of the vector, and therefore takes for argument a single value of type `T` (or convertible to `T`). For multidimensional vectors, this function takes for argument another vector of `D-1` dimensions, and whose lengths match the *last* `D-1` dimensions of the first vector. This new vector is inserted after the existing elements, and the first dimension of the first vector is increased by one.

Example

```
vec1i v = {1,2,3};  
v.push_back(4); // {1,2,3,4}  
  
vec2i w = {{1,2,3}, {4,5,6}};  
w.push_back({7,8,9}); // {{1,2,3}, {4,5,6}, {7,8,9}}  
w.push_back({7,8}); // error: dimensions do not match, 2 != 3
```

- **void** `vec::reserve(std::size_t)`

This function is used for optimization, and is similar to `std::vector::reserve()` (actually, this function is called internally). To understand what this function actually does, one needs to know the internal behavior of `std::vector`. By default, the `std::vector` only allocates enough memory to hold a few elements. Once the allocated memory is full, `std::vector` allocates a larger amount of memory, copies the existing elements inside this new memory, and frees the old memory. This strategy allows virtually unlimited growth of a given vector, and is quite efficiently tuned. However, it is still an expensive operation, and performances can be greatly improved if one knows *in advance* the total number of objects that need to be stored in the vector, so that the right amount of memory is allocated from the beginning, and no copy is required. This function does just that, it tells `std::vector` how many elements it *will* (or might) contain at some point, so that the vector can already allocate enough memory. This is also useful if you only have a rough idea of the future number of elements.

Example

```
vec<int> v = {1,2,3,...};

// Let's imagine you have an algorithm that will produce an
// unknown number of values, but you know that on average
// it is close to N*N.
vec<int> w;
// Reserve roughly enough memory in advance
w.reserve(v.size()*v.size());

// Now the algorithm will run close to the optimal memory
// efficiency
for (...) {
    w.push_back(...);
}
```

- `uint_t` `vec::pitch(uint_t)` **const**

This is only useful for multidimensional vectors. This function returns the “pitch” factor associated to a given dimension. This factor is number of elements in memory that separate two consecutive indices of this dimension. By definition, the pitch factor is 1 for the last dimension. For the other dimensions, this is the product of all

the other dimensions located between the one considered and the last dimension.

Example

```
vec3f v(5,8,6);  
v.pitch(2); // 1  
v.pitch(1); // 6 = v.dims[2]  
v.pitch(0); // 48 = v.dims[2]*v.dims[1]
```

- **bool** `vec::is_same(const vec<D,T>&) const`

This function tests if the provided vector is a view inside this vector.

Example

```
vec1f v = {1,2,3};  
vec1f w = {1,2,3};  
vec1u id = {1,2};  
v.is_same(v[id]); // true  
v.is_same(w[id]); // false
```

- **const** `vec& vec::concretize() const`

This function just returns a reference to this vector, and is only present to mirror the interface of the view class.

- `iterator vec::begin()` and `iterator vec::end()`

These functions allow iteration over the values of this vector. The only reason one may use these functions explicitly is when using algorithms from the standard C++ library, which often work on a pair of iterators as returned by `begin()` and `end()`.

Example

```
vec1f v = {1,2,3};  
// The presence of these functions allow to use vectors  
// in range-based loops  
for (float& f : v) {
```

```
f += 1;
}
```

2.2.4 Indexing

There are two ways to index a given vector, for both mono- and multidimensional vectors. The first way is through “flat” indices and the bracket indexing `v[i]`, i.e., a single index that runs contiguously in memory, and the second way is through multidimensional indices and parenthesis indexing `v(i)` (2.1.2). For monodimensional vectors, these two methods are perfectly identical.

Flat indexing does not care about the details of the dimensions of a given vector. The only important thing is the total number of elements in the vector. This is the simplest and fastest⁵ way to access the data inside a vector.

Example

```
vec3f v(4,5,8); // a complex 3D vector
for (std::size_t i = 0; i < v.size(); ++i) {
    // Here we traverse the vector v regardless of its dimensions
    v[i] = 12.0 + i*i - sqrt(5.0*i) + v[i/4];
}
```

Multidimensional indexing is more involved computationally, because it implies some index arithmetic to compute the flat index and find the right place to read in memory. However it is much more expressive and easier to read and understand. Furthermore, except for very critical code sections, the performance gap is usually negligible, as long as you iterate over the dimensions properly, i.e., following the example below, that you do the *last* nested loop to iterate on the *last* index. This will guarantee as much memory locality as possible, and will take best advantage of CPU caches.

Example

```
vec3f v(4,5,8); // a complex 3D vector
for (std::size_t i = 0; i < v.dims[0]; ++i)
for (std::size_t j = 0; j < v.dims[1]; ++j)
for (std::size_t k = 0; k < v.dims[2]; ++k) {
    // Here we traverse the vector v keeping its dimensional structure
    v(i,j,k) = 42.0 + i + j + k/(j+1);
}
```

⁵Actually there is a faster way using the `safe` wrapper, which does not do bounds checking. See the “Advanced” note at the end of this section.

```
}
```

Indexing a vector can only be done with integers, e.g., `int`, `unsigned int`, `int_t`, `uint_t`, `std::size_t`, etc. Indexing with *unsigned* integers is faster because it removes the need to check for the positivity of the index, and should therefore be preferred when possible. Negative indices are allowed though, and they are interpreted as *reverse* indices, where `-1` refers to the last element of the vector, `-2` the one before the last, etc.

Example

```
vec1i v = {1,2,3,4};
v[0];    // 1, int index
v[0u];   // 1, unsigned int index (faster, but more cumbersome)
vec[-1]; // 4, int index
vec[-2]; // 3, int index
vec[-1] == vec[vec.size()-1]; // always true

v[0.1]; // error: can only access using integers
```

As discussed in the overview (2.1.3), one can also use a vector to index another one. This creates a *view* into the vector. Similarly to scalar indexing, creating a view is only allowed by using integer vectors, i.e., `vec1u`, `vec1i` and their multidimensional counterparts. Again, unsigned integers should be preferred when possible.

Example

```
vec1i v = {1,2,3,4};
vec1u id = {0,2,3};
v[id]; // 1,3,4
```

Sometimes, one will want to use views to access all the elements at once, for example to set all the elements of a vector to a specific value. This can be done with a loop, of course, but it can be boring to write. To do so without explicitly writing the loop, one typically has to create first an index vector containing all the indices of the target vector, and then apply the operation:

```
vec1i v = {1,2,3,4};
vec1u id = {0,1,2,3}; // all the indices of v
v[id] = 12;

// Note that the following will not work
v = 12; // now v only contains a single element equal to 12
```

Not only is this not very practical to write, it is error prone and not very clear. If we decide to add an element to `v`, we also have to modify `id`. Not only this, but it will most likely be slower than writing the loop directly, because the compiler may not realize that you are accessing all the elements contiguously, and will fail to optimize it properly. For this reason, we also introduce the “placeholder” symbol, defined as a single underscore `_`. When used as an index, it means “all the indices in the range”. Coming back to our example:

```
vec1i v = {1,2,3,4};  
v[_] = 12; // it cannot get much shorter
```

This placeholder index can be used in all situations, with both flat and multidimensional indexing. It can be further refined to only encompass a fraction of the whole range, using a peculiar syntax⁶:

```
vec1i v = {1,2,3,4};  
v[_-2] = 12; // only access the indices 0 to 2  
v[2-_] = 12; // only access the indices 2 to 3  
v[1_--2] = 12; // only access the indices 1 to 2
```

Except for the special case of the placeholder index `_`, all the indexing methods described above perform *bound checks* before accessing each element. In other words, the vector class makes sure that each index is smaller than either the total size of the vector (for flat indices) or the length of its corresponding dimension (for multidimensional indices). If this condition is not satisfied, an assertion is raised explaining the problem, and the program is stopped immediately to prevent memory corruption.

Advanced

This bound checking has a small but noticeable impact on performances. In most cases, the added security is definitely worth it. Indeed, accessing a vector with an out of bounds index has very unpredictable impacts on the behavior of the program: sometimes it will crash, but most of the time it will not. Memory will be silently corrupted, the problem will be hard to notice, but the consequences can be terrible... Then, identifying the root of the problem to fix it may prove even more challenging. This is why bound checking is enabled by default.

However, there are cases where bound checking is superfluous, for example if we already know *by construction* that the indices we are dealing with will always be valid. Sometimes the compiler may realize that and optimize the checks away, but one should not rely on it. If these situations are computation-limited, i.e., a lot of time is spent

⁶Be warned that this feature has been introduced recently and may not survive in the future.

doing some number crushing for each element, then the performance hit of bound checking will be negligible, and one should not worry about it. On the other hand, if very little work is done per element, then most of the time will be spent iterating from one index to the next and loading the value in the CPU cache, and bound checking can take a significant amount of the total time.

For this reason, the *phy*₊₊ vector also offers an alternative indexing interface, the *safe* interface, that behaves exactly like the standard interface described above, except that it does not perform bound checking. One may access it using `v.safe[i]` for flat indexing, or `v.safe(x,y)` for multidimensional indexing, and it can also be used to create views. This interface is not meant to be used in daily coding, but rather for computationally intensive functions that you write once but use many times.

2.2.5 Operators

The set of available operators depend on the type of the elements contained in the vector.

- For arithmetic types (**int**, **unsigned int**, **float** and **double**): addition (+), subtraction (-), multiplication (*), division (/) and (integer types only) modulo (%).
- For **bool**: and (&&), or (||) and negation (!).
- For `std::string`: concatenation (+).
- For all types: less than (<), less than or equal (<=), greater than (>), greater than or equal (>=), equal (==) and not equal (!=).

In all cases, the operator overloading allows mixing together vectors of different types (as long as they are convertible one to another) and scalar values.

Example

```
vec<int> v = {1,2,3,4};  
v + 2; // {3,4,5,6}  
v + v; // {2,4,6,8}
```

```
vec<string> fruits = {"apple", "orange"};  
"I ate an "+fruits; // {"I ate an apple", "I ate an orange"}  
2+fruits;           // error: no operator found for int + std::string
```

2.3 The view class

The full type of the view class is

```
template<std::size_t Dimension, typename ElementType>
struct vec<Dimension, ElementType*>;
//           note the asterisk ^
```

The public interface of the view class is very similar to that of the normal vector, and we will not repeat it here. There are some important differences though, which are inherent to the goal of this class. In particular, there is no available constructor (you do now create a view yourself, you ask for it from an existing vector that will create it for you), and the `resize()` function is not available. Lastly, the view implements `concretize()`, differently.

Thanks to their strong similarity, we will not distinguish in the other sections between vectors and views, and will consider views as just another kind of vectors. Indeed, the interface of these two classes has been designed for views to be completely interchangeable with vectors, and vice versa, so that the code of any given function is generally written once and is valid for both types.

2.3.1 Member functions

- `vec<D,T> vec<D,T*>::concretize() const`

This function creates a new vector out of the elements of this view. The returned vector is completely independent from this view, or the original vector this view is currently pointing to. This function is mostly useful when writing generic functions. Indeed, views are implicitly convertible to normal vectors on assignment ([2.2.2](#)).

2.4 Metaprogramming traits and functions (WIP)

Warning

This whole section is more advanced than the rest. It is describing the sets of helper types and metaprogramming functions that one can use to write new functions. It is intended to be followed by readers already familiar with the *phy++* library, and with good knowledge of C++ metaprogramming.

Chapter 3

Support libraries

In this chapter we describe the set of helper functions that are part of the *phy++* support library. These functions are not essential to the use of the *phy++* library, but are mostly modular components that one may choose to use or not. All the support functions are sorted into broad categories to help you discover new functions and algorithm. Alternatively, if you know the name of a function and would like to read its documentation, an index is available at the end of this document.

Note that, in all this section, the signature of the functions is given in pseudo-code, both for conciseness and readability. In particular, the following rules apply.

- The presence of the \textcircled{V} symbol in front of the signature of the function means that this function is also available in a *vectorized* form. This only applies to functions whose first argument is a scalar (i.e., not a vector). In this case, the vectorized form shares the same signature as the original form, but the first argument is promoted to a vector. Calling the vectorized version is the same as writing a loop to call the original version on each element of the vector. If the original function had a return value, the vectorized form returns a vector whose elements are the return value of each call, corresponding to each element of the input vector. The vectorized version can be faster than writing the loop manually.

Example

```
// Suppose this function is marked as vectorized
bool is_odd(uint_t)
// It means that there is another function with the
// same name, but that acts on a vector instead
vec<D, bool> is_odd(vec<D, uint_t>)
```

```

// It is used like this
vec1u v = {1,2,3,4,5};
vec1b b = is_odd(v);
// ... and is equivalent to
vec1b b(v.dims);
for (uint_t i : range(v)) {
    b[i] = is_odd(v[i]);
}

```

- If the function depends on an external library, the name of this library will be written before the signature, for example if LAPACK is needed you will find the symbol [LAPACK].
- Template parameters are not declared explicitly. They are always written in upper-case, usually with a single character (e.g., T, D), or possibly two (e.g., a letter and a number), but never more. Letters T, U, V, etc. refer to template *types*, while letters D, N or I refer to template *integers*.

Example

```

// Pseudo-code used in this section
void foo(T)
// Corresponding C++ code
template<typename T>
void foo(T);

// Pseudo-code used in this section
void foo(vec<D,T> v, U u)
// Corresponding C++ code
template<std::size_t D, typename T, typename U>
void foo(vec<D,T> v, U u);

```

- Template parameters are omitted when not relevant to the description of the function. In this case, it is implicitly assumed that the function will work for any type/value of these template parameters.

Example

```
// Pseudo-code used in this section
void sort(vec&)
// Corresponding C++ code
template<std::size_t D, typename T>
void sort(vec<D,T>&);
```

- There are only two kinds of arguments: input arguments, and input/output arguments. Input arguments are always spelled as plain types, e.g. T, even if the actual signature of the function uses a constant reference, an r-value reference or a universal reference. The reason is that this implementation choice does not matter to the end user. What matters is the interface. The input/output parameters are always C++ references, e.g. T&.

Example

```
// Pseudo-code used in this section
vec1u dims(vec)
// Corresponding C++ code could be either
template<std::size_t D, typename T>
vec1u dims(vec<D,T>);
// ... or
template<std::size_t D, typename T>
vec1u dims(const vec<D,T>&);
// The only difference is that the first version will
// always make a copy (or move) of its parameter, while
// the second may not. This optimization choice depends
// on the actual code inside the function, and has no
// consequence on how the function is actually used.
```

- The ellipsis ... is used to symbolize a list of multiple arguments whose length can vary depending on the context. These arguments are not spelled out explicitly, but the description of the function must make it clear what they are used for. Optionally, a type may be placed before the ellipsis to indicate that all the arguments must be of this same type.

Example

```
// Pseudo-code used in this section
uint_t flat_id(vec, ...)
// Corresponding C++ code
template<std::size_t D, typename T, typename ... Args>
uint_t flat_id(vec<D,T>, Args&& ...);
```

- The std namespace is omitted for common standard types, like std::string or std::array.

Example

```
// Pseudo-code used in this section
uint_t length(string)
// Corresponding C++ code
uint_t length(const std::string&);
```

- For the particular case of std::array, the individual elements inside the array can be named by placing the names inside curly braces after the type of the array.

Example

```
// Pseudo-code used in this section
uint_t distance(array {x,y})
// Corresponding C++ code
template<typename T>
uint_t length(const std::array<T,2>& a) {
    // x := a[0]
    // y := a[1]
}
```

- If the return type of a function in pseudo-code is **auto**, it means that this return value is “complex” (usually a structure or a class) and it is not important to know its precise type. The description of the function must therefore make it clear how this return value can be used.

With this in mind, it is clear that this chapter focuses on the *interface* that is provided by the library, rather than on the individual function themselves. In fact, a single interface may be composed of many different functions to take care of all the combination of types that the interface supports. If the reader is interested in all these overloads, or is experiencing a particular compiler error that cannot be easily fixed just by looking at the interface, then it is best to look directly into the code of the library. Although less readable than the pseudo-code used in this document, most of the time an effort is made to make the code as clear as possible. However, if a function is too hard to understand, I consider this as a bug that should be reported on the `github` issue tracker (seriously). Similarly, if you end up doing something wrong with the library, but that the compiler error message is too cryptic or too long, you may also fill in a bug report. Ensuring that clear error messages are sent to the user is a shared responsibility between compiler writers and library authors.

3.1 Generic vector functions

The vector and view classes are useful and complete tools. However, there are a number of tasks that one repeatedly need to do, like generating a sequence of indices, or sorting a vector, and that would be tedious to write each time they are needed. For this reason, the *phy++* library comes with a large set of utility functions to sort, rearrange, and select data inside vectors. In this section we list these functions and algorithms.

This support library also introduces a global constant called `npos`. This is an unsigned integer whose value is the largest possible integer that the `uint_t` type can hold. It is meant to be used as an error value, or the value to return if no normal value would make sense. It is very similar in concept to the `std::string::npos` provided by the C++ standard library. In particular, it is worth noting that converting `npos` to a *signed* integer produces the value `-1`.

We now describe the functions provided by this library, sorted by categories.

3.1.1 Range-based iteration

- `auto range(vec v)`
`auto range(uint_t n)`
`auto range(uint_t i0, uint_t n)`

This function returns a C++ *range*, i.e., an object that can be used inside the C++ range-based `for` loop. This range will generate integer values starting from `0` (first and second version) or `i0` (third version) to `v.size()` (first version) or `n` (second

and third versions), that last value being *excluded* from the range. This nice way of writing an integer **for** loop actually runs as fast as (if not faster than) the classical way, and is less error prone.

Example

```
vec1i v = {4,5,6,8};

// First version
for (uint_t i : range(v)) {
    // 'i' goes from 0 to 3
    v[i] = ...;
}

// Note that the loop above generates
// *indices* inside the vector, while:
for (int i : v) { /* ... */ }
// ... generates *values* from the vector.

// Second version
for (uint_t i : range(3)) {
    // 'i' goes from 0 to 2
    v[i] = ...;
}

// Third version
for (uint_t i : range(1,3)) {
    // 'i' goes from 1 to 3
    v[i] = ...;
}
```

3.1.2 Index manipulation

- `vec1u mult_ids(vec v, uint_t i)`
`vec1u mult_ids(array<uint_t> dims, uint_t i)`

This is only useful for multidimensional vectors. This function converts a “flat” index `i` ([2.2.4](#)) into an array of multidimensional indices, following the dimensions

of the provided vector `v` (or, in the second version, following the dimensions `dims`). The `flat_id` function does the inverse job.

Example

```
vec2i v(2,3);
mult_ids(v,0); // {0,0}
mult_ids(v,1); // {0,1}
mult_ids(v,2); // {0,3}
mult_ids(v,3); // {1,0}
v[3] == v(1,0); // true
```

- `uint_t flat_id(vec v, ...)`

This is only useful for multidimensional vectors. This function converts a group of multidimensional indices into a “flat” index (2.2.4), following the dimensions of the provided vector `v`. The `mult_ids` function does the inverse job.

Example

```
vec2i v(2,3);
flat_id(v,0,0); // 0
flat_id(v,0,1); // 1
flat_id(v,0,2); // 2
flat_id(v,1,0); // 3
v(1,0) == v[3]; // true
```

3.1.3 Integer sequences

- `vec<D,int_t> indgen(...)`
`vec<D,uint_t> uindgen(...)`
`vec<D,float> findgen(...)`
`vec<D,double> dindgen(...)`

These functions will create a new vector, whose dimensions are specified in argument like in the dimension constructor (2.2.2) or `vec::resize()` (2.2.3). After this vector is created, the function will fill it with values that start at 0 and increment by steps of 1 until the end of the vector.

Example

```
vec1i v = indgen(5);    // {0,1,2,3,4}
vec2u w = uindgen(3,2); // {{0,1}, {2,3}, {4,5}}
```

3.1.4 Rearranging elements

- `vec1u sort(vec)`

void `inplace_sort(vec&)`

These functions will change the order of the elements inside a given vector so that they are sorted from the smallest to the largest. The difference between the two versions is that `sort` does not actually modify the provided vector, but rather returns a vector containing indices inside the provided vector, and `inplace_sort` directly modifies the provided vector. The later is the fastest of the two, but it is less powerful.

Example

```
// First version
vec1i v = {1,5,6,3,7};
vec1u id = sort(v); // {0,3,1,2,4}
v[id]; // {1,3,5,6,7} is sorted
// now, id can also be used to modify the order of
// another vector of the same dimensions

// Second version
inplace_sort(v);
v; // {1,3,5,6,7} is sorted
```

- **bool** `is_sorted(vec)`

This function just traverses the whole input vector and checks if its elements are sorted by increasing value.

Example

```
// First version
vec1i v = {1,5,6,3,7};
is_sorted(v); // false
inplace_sort(v);
v; // {1,3,5,6,7}
is_sorted(v); // true
```

- `vec<1,T> reverse(vec<1,T>)`

This function will inverse the order of all the elements inside the provided vector.

Example

```
vec1i v = {1,2,3,4,5,6};
vec1i w = reverse(v); // {6,5,4,3,2,1}
```

- `vec<1,T> shift(vec<1,T> v, int_t n, T d = 0)`

This function will shift the position of the elements inside the provided vector `v` by a given amount of indices `n`. Elements that would go outside of the bounds of the vector are destroyed. New elements are inserted and default constructed, or assigned the default value `d` (optional argument).

Example

```
vec1i v = {1,2,3,4,5};
vec1i sr = shift(v, 2);
sr; // {0,0,1,2,3}
vec1i sl = shift(v, -2, 99);
sl; // {3,4,5,99,99};
```

3.1.5 Finding elements

- `vec1u where(vec<D,bool>)`

This function will scan the `bool` vector provided in argument, will store the flat indices ([2.2.4](#)) of each element which is `true`, and will return all these indices in a vector. This is a very useful tool to filter and selectively modify vectors, and probably one of the most used function of the whole library.

Example

```
vec1i v = {4,8,6,7,5,2,3,9,0};
// We want to select all the elements which are greater than 3
// We use where() to get their indices
vec1u id = where(v > 3); // {0,1,2,3,4,7}
// Now we can check
v[id]; // {4,8,6,7,5,9}, good!

// The argument of where() can be as complex as you want
id = where(v < 3 || (v > 3 && v % 6 < 2)); // now guess

// It can also involve multiple vectors, as long as they have
// the same dimensions
vec1i w = {9,8,6,1,-2,0,8,5,1};
id = where(v > w || (v + w) % 5 == 0);
// The returned indices are valid for both v and w
v[id]; // {8,6,7,5,2,9}
w[id]; // {8,6,1,-2,0,5}
```

- `vec1u complement(vec v, vec1u id)`

This function works in tandem with `where`. Given a vector `v` and a set of flat indices `id` (2.2.4), it will return the complementary set of indices inside this vector, i.e., all the indices of `v` that are *not* present in `id`.

Example

```
vec1i v = {1,5,6,3,7};
vec1u id = where(v > 4); // {1,2,4}
vec1u cid = complement(v, id); // {0,3}
```

- `uint_t lower_bound(T x, vec v)`
`uint_t upper_bound(T x, vec v)`
`array<uint_t,2> bounds(T x, vec v)`
`array<uint_t,2> bounds(T x1, U x2, vec v)`

These functions use a binary search algorithm to locate the element in the input vector `v` that is equal to or closest to the provided value `x`. It is important to note

that the binary search assumes that the elements in the input vector are *sorted* by increasing value. This algorithm also assumes that the input vector does not contain any NaN value (3.8).

The `lower_bound` function locates the last element in `v` that is less or equal to `x`. If no such element is found, `npos` is returned.

The `upper_bound` function locates the first element in `v` that is greater than `x`. If no such element is found, `npos` is returned.

The first `bounds` function combines what both `lower_bound` and `upper_bound` do, and returns both indices in an array. The second `bounds` function calls `lower_bound` to look for `x1`, and `upper_bound` to look for `x2`.

Example

```
vec1i v = {2,5,9,12,50};
bounds(0, v);    // {npos,0}
bounds(9, v);    // {2,3}
bounds(100, v);  // {4,npos}
```

- `vec1u equal_range(T x, vec v)`

Similarly to `lower_bound`, `upper_bound` and `bounds`, this function uses a binary search algorithm to locate all the values in the input vector `v` that are equal to `x`. It then returns the flat indices of all these values in a vector. It is important to note that the binary search assumes that the elements in the input vector are *sorted* by increasing value. This algorithm also assumes that the input vector does not contain any NaN value (3.8).

If no such value is found, an empty vector is returned.

Example

```
vec1i v = {2,2,5,9,9,9,12,50};
equal_range(9, v); // {3,4,5}
```

```
// It's a faster version of
where(v == 9);
```

- `vec1u uniq(vec v)`


`vec1u uniq(vec v, vec1u sid)`

This function will traverse the provided vector `v` and find all the unique values. It will store the indices of these values (if a value is present more than once inside `v`, the index of the first one will be used) and return them inside an index vector. The first version assumes that the values in `v` are *sorted* from the smallest to the largest. In the second version, `v` may not be sorted, but the second argument `id` contains indices that will sort `v` (e.g., `id` can be the return value of `sort(v)`).

Example

```
// For a sorted vector
vec1i v = {1,1,2,5,5,6,9,9,10};
vec1u u = uniq(v); // {0,2,3,5,6,8}
v[u]; // {1,2,5,6,9,10} only unique values
```

```
// For an non-sorted vector
vec1i w = {5,6,7,8,6,5,4,1,2,5};
vec1u u = uniq(w, sort(w));
w[u]; // {1,2,4,5,6,7,8}
```

-  **bool** `is_any_of(T1 v1, vec<D2,T2> v2)`

This function looks if there is any value inside `v2` that is equal to `v1`. If so, it returns **true**, else it returns **false**.

Example

```
vec1i v = {7,4,2,1,6};
vec1i d = {5,6,7};
vec1b b = is_any_of(v, d); // {true, false, false, false, true}
```

- **void** `match(vec v1, v2, vec1u& id1, id2)`

This function traverses `v1` and, for each value in `v1`, looks for elements in `v2` that have the same value. If one is found, the flat index of the element of `v1` is added to `id1`, and the flat index of the element of `v2` is added to `id2`. Then the function goes on to the next value in `v1`. Note that, contrary to `match_dictionary`, each value

in v_1 is matched to *at most* one value in v_2 , and vice versa. In other words, if v_1 or v_2 contain duplicates, only the first value will be matched and the others will be ignored. This function is symmetric: the result will be the same if you swap the two input vectors (of course, the output vectors have to be swapped also).

Example

```
vec1i v = {7,4,2,1,6};
vec1i w = {2,6,5,3};
vec1u id1, id2;
match(v, w, id1, id2);
id1; // {2,4}
id2; // {0,1}
v[id1] == w[id2]; // always true
```

- **void** match_dictionary(vec v1, v2, vec1u& id1, id2)

This function traverses v_1 and, for each value in v_1 , looks for an element in v_2 that has the same value. If one is found, the flat index of the element of v_1 is added to id_1 , and the flat index of the element of v_2 is added to id_2 . Then the function goes on to the next value in v_1 . Contrary to `match`, each value in v_2 can be matched to multiple values in v_1 . Therefore this function is not symmetric: the second vector should be considered as a “dictionary”, hence the name of this function. It is assumed that v_2 does not contain any duplicates.

Example

```
vec1i v = {7,6,2,1,6};
vec1i w = {2,6,5,3};
vec1u id1, id2;
match_dictionary(v, w, id1, id2);
id1; // {1,2,4}
id2; // {1,0,1}
v[id1] == w[id2]; // always true
```

- **bool** astar_find(vec2b m, uint_t& x, y)

This function uses the A^* (“A star”) algorithm to look inside a 2D boolean map m and, starting from the position x and y (i.e. $m(x,y)$), find the closest point that has a

value of **true**. Once this position is found, its indices are stored inside `x` and `y`, and the function returns **true**. If no element inside `m` is **true**, then the function returns **false**.

Example

```
// Using '@' for true and '.' for false,
// assume we have the following boolean map,
// and that we start at the position indicated
// by 'S', the closest point whose coordinates
// will be returned by astar_find is indicated
// by an 'X'
vec2b m;
// .....
// .....
// .....
// ...aaaaa.....
// ...aaaaa.....
// ...aaaaa.....
// ...aaaaa.....
// ...aaaaa.....
// ...aaaaX.....
// .....S....
// .....
// .....

uint_t x = 13, y = 2;
astar_find(m, x, y);
x; // 7
y; // 3
```

3.1.6 Modifying dimensions

- `vec<1,T> flatten(vec<D,T>)`

This function transforms a multidimensional vector into a monodimensional vector. The content in memory is exactly the same, so the operation is fast. In particular, if the argument of this function is a temporary, this function is extremely cheap as it produces no copy. The `reform` function does the inverse job, and more.

Example

```
vec2i v = {{1,2,3}, {4,5,6}};  
vec1i w = flatten(v); // {1,2,3,4,5,6}
```

- `vec<D1,T> reform(vec<D2,T>, ...)`

This function transforms a vector into another vector just by changing its dimensions. The content in memory is exactly the same, so the operation is fast. In particular, if the argument of this function is a temporary, this function is extremely cheap as it produces no copy. However, the new dimensions have to generate the same number of elements as the provided vector contains. The `flatten` function is a special case of `reform`.

Example

```
vec1i v = {1,2,3,4,5,6};  
vec2i w = reform(v, 2, 3); // {{1,2,3}, {4,5,6}}
```

- `vec<N,T> replicate(T, ...)`
`vec<D+N,T> replicate(vec<D,T>, ...)`

This function will take the provided scalar (first version) or vector (second version), and replicate it multiple times according to the provided additional parameters, to generate additional dimensions.

Example

```
// First version  
vec1i v = replicate(2, 5);  
v; // {2,2,2,2,2} 5 times 2  
vec2i w = replicate(2, 3, 2);  
w; // {{2,2},{2,2},{2,2}} 3 x 2 times 2  
  
// Second version  
vec2i z = replicate(vec1i{1,2}, 3);  
z; // {{1,2},{1,2},{1,2}} 3 times {1,2}
```

```
// Note that it is not possible to just use a plain initializer list
vec2i z = replicate({1,2}, 3); // error, unfortunately
```

3.1.7 Adding/removing elements

- `vec<1,T> remove(vec<1,T> v, vec1u id)`

void `inplace_remove(vec<1,T>& v, vec1u id)`

These functions will remove the elements in `v` that have their indices in `id`. The only difference between the first and the second version is that the former will first make a copy of the provided vector, remove elements inside this copy, and return it. The second version modifies directly the provided vector, and is therefore faster.

Example

```
// First version
vec1i v = {4,5,2,8,1};
vec1i w = remove(v, {1,3}); // {4,2,1}

// Second version
inplace_remove(v, {1,3});
v; // {4,2,1}
```

- **void** `append<N>(vec<D,T1>& t1, vec<D,T2> t2)`
void `append(vec<1,T1>& t1, vec<1,T2> t2)`
void `prepend<N>(vec<D,T1>& t1, vec<D,T2> t2)`
void `prepend(vec<1,T1>& t1, vec<1,T2> t2)`

These functions behave similarly to `vec::push_back`, in that they will add new elements at the end (`append`), but also at the beginning (`prepend`) of the provided vector. However, when `vec::push_back` can only add new elements from a vector that is one dimension less than the original vector (or a scalar, for monodimensional vectors), these functions will add new elements from a vector of the *same* dimension. These functions are also more powerful than `vec::push_back`, because they allow you to choose along which dimension the new elements will be added, through the template parameter `N` (note that this parameter is useless and therefore does not exist for monodimensional vectors). The other dimensions must be identical. They are clearly dedicated to a more advanced usage.

Example

```
// For monodimensional vectors
vec1i v = {1,2,3};
vec1i w = {4,5,6};
append(v, w);
v; // {1,2,3,4,5,6}
prepend(v, w);
v; // {4,5,6,1,2,3,4,5,6}

// For multidimensional vectors
vec2i x = {{1,2}, {3,4}}; // x is (2x2)
vec2i y = {{0}, {0}}; // y is (1x2)
vec2i z = {{5,6,7}}; // z is (3x1)
append<1>(x, y);
x; // {{1,2,0}, {3,4,0}} // x is (2x3)
prepend<0>(x, z);
x; // {{5,6,7}, {1,2,0}, {3,4,0}} // x is (3x3)
```

3.1.8 Error checking

- **void** phypp_check(**bool** b, ...)

This function makes error checking easier. When called, it checks the value of b. If b is **true**, then nothing happens. However if b is **false**, then the current file and line are printed to the standard output, followed by the other arguments of this function (they are supposed to be an error message explaining what went wrong), and an assertion is raised, immediately stopping the program. This function is used everywhere in the *phy++* library to make sure that certain conditions are properly satisfied before doing a calculation, and it is essential to make the program crash in case something is unexpected (rather than letting it run hoping for the best, and often getting the worst).

Since this function is actually implemented by a preprocessor macro, you should not worry about its performance impact. It will only affect the performances when something goes wrong and the program is about to crash. However the calculation of b can itself be costly (for example, you may want to check that a vector is sorted), and there is no way around this.

Example

```
vec<T> v;
// Suppose v is read from the command line arguments.
v = read_from_somewhere();

// The following code needs at least 3 elements in the
// vector v, so we need to check that first.
phypp_check(v.size() >= 3, "this algorithm needs at least "
            "3 values in the input vector, but only ", v.size(),
            " were found");

// If we get past this point, then v has the right number
// of elements, and we can proceed
do_stuff(v);
```

3.2 Parsing command line arguments

- **void** read_args(**int** argc, **char*** argv[], ...)

The biggest problem of C++ programs is that, while they are usually fast to execute, they can take a long time to *compile*. For this reason, C++ is generally not a productive language in situations where the code has to be written by *trial and error*, a process that involves frequently changing the behavior or starting point of a program. This situation could change the day we have a complete, robust and efficient C++ interpreter (keep an eye out for **cling**).

Fortunately, in the mean time there are ways around this issue. One in particular is called *data driven* programming: the behavior of a program depends on the data that are fed to it. The simplest way to use this paradigm is to control the program through *command line arguments*. The C++ language provides the basic bricks to use command line arguments, but the interface is inherited from C and lacks severely in usability. For this reason we introduce in *phy++* a single function `read_args()` that uses these bricks to provide a simple and concise interface to implement command line arguments in a program. The details of the implementation are complex, since this function makes use of preprocessor macros, so instead of describing it all, we will focus on a simple example.

Let us assume that we want to build a simple program that will print (3.6) to the terminal the first *n* powers of two.

Example

```
// This is the standard entry point of every C++ program.
// The signature of the main function is imposed by the C++ standard
int main(int argc, char* argv[]) {
    // Declare the main parameters of the program

    // The number of powers of two to display
    uint_t n = 1;

    // Then read command line arguments...
    read_args(argc, argv, arg_list(n));

    // - The first two arguments must be the arguments of the main
    //   function, in the exact same order.
    // - The following argument must be arg_list(...).

    // Within the arg_list(), one can put as many C++ variables as
    // needed. The function will recognize their name, and will try
    // to find a command line argument that matches.
    // If it finds one, it tries to convert its value into the type
    // of the variable, and if successful, store this new value
    // inside the variable. In all other cases, the variable is not
    // modified.
    // It is therefore important to give a meaningful default value
    // to each variable!

    // Now we can go on with the program
    print(pow(2.0, findgen(n)+1));

    // And quit gracefully
    return 0;
}
```

By just adding this line

```
read_args(argc, argv, arg_list(n));
```

we exposed the variable `n` to the public: everyone that runs this program can modify the value of `n` to suit their need. Assuming the name of the compiled program is `show_pow2`, then the program is ran the following way:

```

# First try with no parameter.
# 'n' is not modified, and keeps its default value of 1.
./show_pow2
# output:
# 2

# Then we change 'n' to 5.
./show_pow2 n=5
# output:
# 2 4 8 16 32

```

You immediately see the interest of this approach. Instead of recompiling the whole program just to change `n`, we expose it in the program arguments. We can then compile the program once, and change its behavior without recompiling. This saves a lot of time when trying to figure out what is the best value of a parameter in a given problem, or if we have created an algorithm that depends on some parameters, and we want to see how the results change when we change the values of these parameters. And of course, this is most useful when writing *tools* (see, e.g., the tools presented in Chapter 4).

In the previous example, we chose to expose a simple integer. But in fact, the interface also allows virtually any type, provided that there is a way to convert a string into a value of this type. In particular, this is the case for vectors. There is a little subtlety though: the values of the vector must be separated by commas, *without any space* (unless you put the whole argument inside quotes), and surrounded by brackets `[...]`. Again, let us illustrate this with an example. We will modify the previous program to allow it to show not only powers of 2, but the powers of multiple, arbitrary numbers. Note: in the following, we will not repeat the whole `main()` function, just the important bits.

Example

```

// The number of powers of two to display
uint_t n = 1;
// The powers to display
vec1f p = {2};

// Read command line arguments
read_args(argc, argv, arg_list(n, p));

```

```
// Go on with the program
for (float v : p) {
    print(pow(v, findgen(n)+1));
}
```

The program can now change the powers it displays, for example

```
# We keep 'n' equal to 5, and we show the powers of 2, 3 and 5.
./show_pow2 n=5 p=[2,3,5]
# output:
# 2 4 8 16 32
# 3 9 27 81 243
# 5 25 125 625 3125
```

Finally, you may think that `p` is not a very explicit name for this last parameter. It would be clearer if we could call it `pow`. Unfortunately, `pow` is already the name of a function in C++, so we cannot give this name to the variable. However, the `read_args()` interface allows you to manually give a name to any parameter using the `name()` function. Let us do that and modify the previous example.

Example

```
// The number of powers of two to display
uint_t n = 1;
// The powers to display, we still call it 'p' in the program
vec1f p = {2};

// Read command line arguments
read_args(argc, argv, arg_list(n, name(p, "pow")));

// Go on with the program
for (float v : p) {
    print(pow(v, findgen(n)+1));
}
```

Now we will call instead

```
./show_pow2 n=5 pow=[2,3,5]
# output:
# 2 4 8 16 32
```

```
# 3 9 27 81 243
# 5 25 125 625 3125
```

3.3 String manipulation

3.3.1 String conversions

- `string strn(T v)`

```
string strn_sci(T v)
```

These functions will convert the value `v` into a string. This value can be of any type, as long as it is convertible to string. In particular, `v` can be a vector, in which case the output string will contain all the values of the vector, separated by commas, and enclosed inside curly braces `"{...}"`.

Example

```
strn(2);           // "2"
strn(true);        // "1"
strn("foo");        // "foo"
strn(vec1i{2,5,9}); // "{2, 5, 9}"
```

The second version is dedicated to floating point numbers, and will output them in scientific format.

Example

```
strn_sci(2.0);     // "2.000000e+00"
strn_sci(2e10);    // "2.000000e+10"
strn_sci(-5e-2);   // "-5.000000e-02"

// Integer numbers are not affected
strn_sci(1000);    // "1000"
strn_sci(1000.0);  // "1.000000e+03"
```

- `vec_<D,string> strna(vec<D,T>)`
`vec_<D,string> strna_sci(vec<D,T> v)`

These functions are the vectorized version of `strn`. The reason why the name of the function is different is because it is already possible to call `strn` with vector arguments: the whole vector will be converted into a *single* string. These versions however convert each element separately to form a vector of strings.

Example

```
strna(vec1i{1,5,6,9}); // {"1", "5", "6", "9"}
strn(vec1i{1,5,6,9});  // "{1, 5, 6, 9}"
```

- **ⓧ** **bool** `from_string(string s, T& v)`

This function tries to convert the string `s` into a C++ value `v` and returns **true** in case of success. If the string cannot be converted into this value, for example if the string contains letters and the value has an arithmetic type, or if the number inside the string is too big to fit inside the C++ value, the function will return **false**. In this case, the value of `v` is undefined.

The vectorized version of this function will try to convert every value inside the string vector `s`, and will store the converted values inside the vector `v` (it will take care of properly resizing the vector, so you can pass an empty vector if you want). The return value will then be an array of boolean values, corresponding to the success or failure of each individual value inside `s`. If an element of `s` fails to convert, the corresponding value in `v` will be undefined.

Example

```
float f;
bool b = from_string("3.1415", f);
b; // true
f; // 3.1415

b = from_string("abcdef", f);
b; // false;
f; // ??? could be 3.1415, or NaN, or anything else

vec1f fs;
vec1b bs = from_string({"1", "1.00e5", "abc", "1e128", "2.5"}, fs);
bs; // {true, true, false, false, true}
```

```
fs; // {1, 1e5, ???, ???, 2.5}
```

- `string hash(...)`

This function scans all the arguments that are provided, and returns the hexadecimal representation of the SHA-1 hash of this argument list. The hash is a string such that: 1) all further calls of `hash(...)` with arguments that have the exact same value will always return the same string, and 2) the probability that the function returns the same string for another set of arguments is very small (in fact, although this algorithm was created in 1995, no such collision has been found yet).

This is useful for example to cache the result of some heavy computation: once the computation is done, the input parameters of the computation can be fed to `hash()` to give a “sort-of-unique” identifier to the result, and this result can be saved somewhere with this identifier. Later on, if the computation is requested with another set of parameters, they are fed to `hash()` and the resulting string is compared to all the saved results: if a match is found, then the associated pre-computed result can be used immediately, else the computation goes on.

Example

```
std::string s;
```

```
// With a single argument
s = hash("hello world!");
s; // "da52a1357f3c973e1ffc1b694d5308d0abcd9845"
s = hash("hello world?");
s; // "793e673d04e555f8f0b38033d5223c525a040719"
// Notice how changing a single character gives a completely
// different hash string
```

```
// With multiple arguments
s = hash(1, 2, 3);
s; // "570331ab965721aae8a8b3c628cae57a21a37560"
s = hash("123");
s; // "0e898437b29ec20c39ca48243e676bcb177d4632"
s = hash(1.0, 2.0, 3.0);
s; // "9c45014f7c7943cb7860f3db4b885fb44b510ec8"
```

3.3.2 String operations

- ① **bool** `empty(string s)`

This function will return **true** if the provided string does not contain any character *at all* (including spaces), and **false** otherwise.

Example

```
vec1s str = {"", "abc", "  "};
vec1b b = empty(str);
b; // {true, false, false}
// Not to be confused with the vec::empty() function
str.empty(); // false
str = {" "};
str.empty(); // false
str = {};
str.empty(); // true
```

- ① **uint_t** `length(string s)`

This function will return the length of the provided string, i.e., the number of character it contains (including spaces). If the string is empty, the function will return zero.

Example

```
vec1s str = {"", "abc", " a b"};
vec1u n = length(str);
n; // {0, 3, 4}
```

- ① **string** `trim(string s, c = " \t")`

This function will look at the beginning and the end of the string `s` and search for any of the characters that is present in `c`. If one is found, then this character is removed from `s`. This procedure is repeated until no character is found. The net effect of this function is that the provided string `s` is *trimmed* from any of the characters listed in `c`. This is useful for example to remove leading and trailing spaces of a string (default value of `c`), or to removes quotes.

Example

```
vec1s str = {"", "abc", " a b", " a b c "};
vec1s t = trim(str, " "); // trim spaces
t; // {"", "abc", "a b", "a b c"}

str = {"", "(a,b)", "((a,b),c)"};
t = trim(str, "()"); // trim parentheses
t; // {"", "a,b", "a,b),c"}
```

- ④ string toupper(string s)

This function will transform all characters of the string to be upper case. It has no effect on non-alphabetic characters such as numbers, punctuation, of special characters.

Example

```
vec1s str = {"", "abc", "AbCdE", "No, thanks!"};
vec1s t = toupper(str);
t; // {"", "ABC", "ABCDE", "NO, THANKS!"}
```

- ④ string tolower(string s)

This function will transform all characters of the string to be lower case. It has no effect on non-alphabetic characters such as numbers, punctuation, of special characters.

Example

```
vec1s str = {"", "abc", "AbCdE", "No, thanks!"};
vec1s t = tolower(str);
t; // {"", "abc", "abcde", "no, thanks!"}
```

- ④ string replace(string s, p, r)

This function will look inside the string s for occurrences of p and replace each of them with r. The string is unchanged if no occurrence is found. In particular, this

function can be used to remove all the occurrences of `p` simply by setting `r` equal to an empty string.

Example

```
vec1s str = {"I eat apples", "The apple is red"};
vec1s r = replace(str, "apple", "pear");
r; // {"I eat pears", "The pear is red"}

str = {"a:b:c", "g:p"};
r = replace(str, ":", ",");
r; // {"a,b,c", "g,p"}
```

- `vec1s split(string s, p)`

This function will split the string `s` into a vector of sub-strings each time the pattern `p` occurs. If no such pattern is found in `s`, the function returns a vector containing a single element which is the whole string `s`.

Example

```
vec1s str = split("this is the end", " ");
str; // {"this", "is", "the", "end"};
str = split("a:b::c:d");
str; // {"a", "b", "", "c", "d"}
```

- `vec1s cut(string s, uint_t n)`

This function will split the string `s` into a vector of sub-strings (or “lines”) that are exactly `n` characters long (except possibly the last one). Contrary to the function `wrap()`, this function does not care about spaces and preserving the boundaries of words.

Example

```
vec1s str = cut("this is the end", 5);
str; // {"this ", "is th", "e end"}
```

- `vec1s wrap(string s, uint_t w, string i = "", bool e = false)`

This function will split the string `s` into a vector of sub-strings (or “lines”) that are at most `w` characters long. Contrary to the function `cut()`, this function takes care of not splitting words into multiple parts. If a cut would happen in the middle of a word, then the cut is shifted back to the beginning of the word, and the latter is flushed to the next line. If a word is larger than `w`, then it will be alone on its line. Alternatively, in such a case if `e` is set to `true`, the word is truncated and the last characters are lost. An ellipsis `"..."` is also appended to notify that the word has been truncated. Finally, the parameter `i` can be used to add indentation: these characters are added at the beginning of the line and are taken into account when calculating the length of the line. In this case the first line is not indented, to allow using a different header. This function is useful to display multi-line messages on the terminal.

Example

```
std::string str = "This is an example text with many words. Just "
                 " for the sake of the example, we are going to write a "
                 "veryyyyyyyyyyyyyyyyyyyyyyyyy long word.";
```

```
vec1s s = wrap(str, 23);
s[0]; // "This is an example text"
s[1]; // "with many words. Just "
s[2]; // "for the sake of the"
s[3]; // "example, we are going"
s[4]; // "to write a"
s[5]; // "veryyyyyyyyyyyyyyyyyyyyyyyyy"
s[6]; // "long word."
```

```
vec1s s = wrap(str, 23, "", true);
s[0]; // "This is an example text"
s[1]; // "with many words. Just "
s[2]; // "for the sake of the"
s[3]; // "example, we are going"
s[4]; // "to write a"
s[5]; // "veryyyyyyyyyyyyyyyyyyyy..."
s[6]; // "long word."
```

```
vec1s s = wrap(str, 23, " ", true);
s[0]; // "This is an example text"
```

```
s[1]; // " with many words. Just"
s[2]; // " for the sake of the"
s[3]; // " example, we are going"
s[4]; // " to write a"
s[5]; // " veryyyyyyyyyyyyyyyyyyy..."
s[6]; // " long word."
```

- `string collapse(vec<D,string> v, string s = "")`

This function will concatenate together all the strings present in the vector `v` to form a single string. A separator can be provided using the argument `s`, in which case the string `s` will be inserted between each pair of strings of `v` that are to be concatenated.

Example

```
vec1s v = {"a", "b", "c"};
std::string s = collapse(v);
s; // "abc"

s = collapse(v, ", ");
s; // "a, b, c"
```

- ④ `uint_t find(string s, p)`

This function returns the position in the string `s` of the first occurrence of the substring `p`. If no such occurrence is found, the function returns `npos`.

Example

```
vec1s v = {"Apple", "please", "complementary", "huh?"};
vec1u p = find(v, "ple");
p; // {2, 0, 3, npos}
```

- ④ `bool regex_match(string s, r)`
- ④ `bool regex_match_any_of(string s, vec1s r)`

This function will return `true` if the string `s` matches the regular expression (regex) `r`. This regular expression can be used to identify complex patterns in a string,

far more advanced than just matching the existence of a sub-string. This particular implementation uses POSIX regular expressions. The syntax is complex and not particularly intuitive, but it has become a well known standard and is therefore understood by most programmers. A gentle tutorial can be found [here](#). Note that if the regular expression is invalid, an error will be reported to diagnose the problem, and the program will stop.

The function `regex_match_any_of` will do the same, but for a list of regular expressions. `true` will be returned if the string `s` matches at least one of these.

Lastly, it is advised to use *raw string literals* to specify the regex in C++ code. Indeed, one often needs to use the backslash character (`\`), but this character is used in C++ to create escape sequences like `'\n'` (new line). For this reason, to feed the backslash to the regex compiler you actually need to escape it: `"\\"`. To avoid this inconvenience, just enclose the regular expression in `R"(...)"`, and you will not need to worry about escaping characters from the C++ compiler. Note however that you will still need to escape characters from the regular expression compiler itself, when needed.

Example

```
vec1s v = {"abc,def", "abc:def", "956,fgt", "9g5,hij", "ghji,abc"};
```

```
// We want to find which strings have the "XXX,YYY" format
// where "XXX" can be any combination of three letters or numbers
// and "YYY" can be a combination of three letters
vec1b b = regex_match(v, R"([a-z0-9]{3},[a-z]{3}$)");
```

```
// This regular expression can be read like:
// '^'   : at the beginning of the string, match...
// '['   : any character among...
// 'a-z' : letters from 'a' to 'z'
// '0-9' : numbers from 0 to 9
// ']'   :
// '{3}' : three times
// ','   : followed by a coma, then...
// '['   : any character among...
// 'a-z' : letters from 'a' to 'z'
// ']'   :
// '{3}' : three times
// '$'   : and then the end of the string
```

```

b[0]; // true
b[1]; // false, there is no ","
b[2]; // true
b[3]; // true
b[4]; // false, too many characters before the ","

```

- `vec2s regex_extract(string s, r)`

This function will analyze the string `s`, perform regular expression matching (see `regex_match`) using the regular expression `r`, and will return a vector containing all the extracted substrings. To extract one or more substrings in the regular expression, just enclose the associated patterns in parentheses. The returned vector is two dimensional: the first dimension corresponds to the number of times the whole regular exception was matched in the provided string, the second dimension corresponds to each extracted substring.

Example

```

std::string s = "array{5,6.25,7,28}end45.6ddfk 3.1415";

// We want to find all floating point numbers in this mess, and
// extract their integer and fractional parts separately.
vec2s sub = regex_extract(s, R"(([0-9]+)\.([0-9]+))");

// The regular expression can be read like:
// '(' : open a new sub-expression containing...
// '[' : any character among...
// '0-9' : the numbers 0 to 9
// ']' : end of the sub-expression
// '+' : with at least one such character
// ')' : end of the sub-expression
// '\.' : followed by a dot (has to be escaped with '\')
// '(' : open a new sub-expression containing...
// ... exactly the same pattern as the first one
// ')' : end of the sub-expression

// So we are looking for two sub-expressions, the first is the
// integral part of the floating point number, and the second is the
// fractional part.

```

```
// It turns out that there are three locations in the input string
// that match this pattern:
sub(0,_); // {"6", "25"}
sub(1,_); // {"45", "6"}
sub(2,_); // {"3", "1415"}
```

- `string regex_replace(string s, reg, T fun)`

This function will search in the string `s` using the regular expression `reg` (see `regex_match`) to locate some expressions. Parts of these expressions can be *captured* by enclosing them in parenthesis. These captured sub-expressions are extracted from `s`, stored inside a string vector, and fed to the user-supplied “replacement function” `fun`. In turn, this function can analyze and modify the captured sub-expressions and produce a new replacement string that will be inserted in place of the matched expression.

If no sub-expression is captured, then the string vector that is fed to `fun` will be empty. Expressions are found and replaced in the order in which they appear in the input string `s`.

This function is very similar to the `sed` program.

Example

```
std::string s = "a, b, c=5, d=9, e, f=2, g";

// First a simple example.
// We want to find all the "X=Y" expressions in this string and
// add parentheses around "Y".
std::string n = regex_replace(
    s, // the string to analyze
    R"(([a-z])=([0-9]))", // the regular expression
    // the replacement function
    [](vec1s m) {
        // "X" is in m[0], "Y" is in m[1].
        return m[0] + "(" + m[1] + ")";
    }
);

// The regular expression can be read like:
// '(' : open a new sub-expression containing...
// '[' : any character among...
```

```

// 'a-z' : the letters 'a' to 'z'
// ']'
// ')' : end of the sub-expression
// '=' : followed by the equal sign
// '(' : open a new sub-expression containing...
// '[' : any character among...
// '0-9' : the numbers 0 to 9
// ']'
// ')' : end of the sub-expression

// The result is:
n; // "a, b, c=(5), d=(9), e, f=(2), g"


// A second, more complex example.
// We take the same example as above, but this time we want to
// change "X" to upper case and increment "Y" by one.
std::string n = regex_replace(
    s, // the string to analyze
    R"(([a-z])=([0-9]))", // the regular expression
    // the replacement function
    [](vec1s m) {
        // Again, "X" is in m[0], "Y" is in m[1].


        // Read the integer "Y" and increment it
        uint_t k;
        from_string(m[1], k);
        ++k;

        // Build the replacement string
        return toupper(m[0])+"="+strn(k);
    }
);

// The result is:
n; // "a, b, C=6, D=10, e, F=3, g"

```

-  **bool** start_with(string s, string p)

-  **bool** end_with(string s, string p)





These functions will return **true** if the beginning (start_with) or the end (end_with)

of the string *s* is *exactly* the same as the string *p*.

Example

```
vec1s v = {"p1_m2.txt", "p3_c4.fits", "p1_t8.fits"};
vec1b b = start_with(v, "p1");
b; // {true, false, true}

b = end_with(v, ".fits");
b; // {false, true, true}
```

-  string erase_begin(string s, string p)
-  string erase_begin(string s, uint_t n)
-  string erase_end(string s, string p)
-  string erase_end(string s, uint_t n)

These functions will erase a number of characters from the beginning (erase_begin) or the end (erase_end) of the string *s*. If the second argument is a string *p*, then the function first checks that the string begins/ends with *p*: if yes, it removes this substring from *s*; if no, an error is reported and the program stops. If the second argument is a number *n*, then *n* characters are removed. If *n* is larger than the size of *s*, the returned string will be empty.

Example



```
vec1s v = {"p1_m2.txt", "p3_c4.fits", "p1_t8.fits"};
std::string s = erase_begin(v[0], "p1_");
s; // "m2.txt"
s = erase_begin(v[1], "p1_");
// will trigger an error
s = erase_begin(v[2], "p1_");
s; // "t8.fits"

s = erase_end(v[0], ".fits");
// will trigger an error
s = erase_end(v[1], ".fits");
s; // "p3_c4"
s = erase_end(v[2], ".fits");
```



```
s; // "p1_t8"




vec1s t = erase_begin(v, 3);
t; // {"m2.txt", "c4.fits", "t8.fits"}
t = erase_end(v, 5);
t; // {"p1_m", "p3_c4", "p1_t8"}
```

-  `string keep_start(string s, uint_t n = 1)`
-  `string keep_end(string s, uint_t n = 1)`

These functions will return the first (`keep_start`) or last (`keep_end`) `n` characters of the string `s` and discard the rest. If `n` is larger than the size of `s`, the whole string is returned untouched.

Example

```
vec1s v = {"p1_m2.txt", "p3_c4.fits", "p1_t8.fits"};
vec1s s = keep_start(v, 2);
s; // {"p1", "p3", "p1"}
s = keep_end(v, 4);
s; // {".txt", ".fits", ".fits"}
```

-  `string align_left(string s, uint_t w, char f = ' ')`
-  `string align_center(string s, uint_t w, char f = ' ')`
-  `string align_right(string s, uint_t w, char f = ' ')`

These functions will pad the provided string with the character `f` (default to a space) so that the total width the returned string is equal to `w`. If the provided string is larger than `w`, it is returned untouched. Padding characters will be appended at the end of the string (`align_left`), at the beginning of the string (`align_right`), or both (`align_center`).

Example

```
std::string s = "5.0";
std::string n = align_left(s, 6);
n; // "5.0  "
n = align_right(s, 6);
```

```

n; // " 5.0"
n = align_center(s, 6);
n; // " 5.0 "

// Another padding character can be used
n = align_left(s, 6, '0');
n; // "5.0000"

```

- **④** `uint_t distance(string s1, s2)`

This function computes the *lexicographic distance* between two strings. The definition of this distance is the following. If the two strings that are exactly identical, the distance is zero. Else, each character of the shortest string are compared to their equivalent at the same position in the other string: if they are different, the distance is increase by one. Finally, the distance is increased by the difference of size between the two strings.

The goal of this function is to identify *near* matches in case a string could not be found in a pre-defined list. This is useful to suggest corrections to the user, who may have misspelled it.

Example

```

vec<string> s = {"wircam_K", "hawki_Ks", "subaru_B"};
vec<uint_t> d = distance(s, "wirkam_Ks");
d; // {2, 8, 8}

// Nearest match
std::string m = s[min_id(d)];
m; // "wircam_K"

```

- `string replace_block(string v, b, e, T f)`

`string replace_blocks(string v, b, s, e, T f)`

The `replace_block` function will look in the string `v` and identify all blocks that start with `b` and end with `e`. The content of each block is fed to the user-supplied function `f` which does any kind of conversion or operation, and returns a new string. This new string is inserted in `v` in place of the block.

The `replace_blocks` function will do the same thing, except that each block can have multiple “arguments” that are separated with `s`. In this case, the function extracts

all the “arguments” and stores them inside a string vector, and feeds this vector to the conversion function `f`.

Example

```
// We want to modify the content inside <b>...</b> to be upper case
std::string s = "This is a <b>whole</b> lot of <b>money</b>";
std::string ns = replace_block(s, "<b>", "</b>", [](std::string t) {
    return toupper(t);
});

ns; // "This is a <b>WHOLE</b> lot of <b>MONEY</b>"

// We want to convert this LaTeX link into HTML
s = "Look at \url{http://www.google.com}{this} link.";
ns = replace_blocks(s, "\url{", "{", "}", [](vec1s t) {
    return "<a href=\""+t[0]+"\">"+t[1]+"</a>";
});

ns; // "Look at <a href=\"http://www.google.com\">this</a> link."
```


3.4 OS interaction

- `string system_var(string v, string d = "")`

3.5 File input/output

3.5.1 File system

This section describes the function provided by the `phy++` library that are related to file system management, i.e., getting informations, creating or copying files and directories. All paths can be either absolute, or relative to the working directory.

-  `bool file::exists(string)`

This function returns `true` if the path given in argument corresponds to a file or directory that actually exists on the disk, and `false` otherwise.

Example

```
bool b = file::exists("~/phypprc");  
b; // hopefully true  
  
b = file::exists("/i/do/not/exist");  
b; // probably false
```

- **bool** file::is_older(string f1, string f2)

This function returns **true** if the file or directory f1 is *older* than the file or directory f2. The age of a file corresponds to the time spent since the files were last modified. If one of the two files does not exist, the function returns **false**.

Example

```
bool b = file::is_older("~/phypprc", "/usr/bin/cp");  
b; // probably false
```

- **vec1s** file::list_directories(string)

This function scans the directory given in argument and returns the list of all directories it contains. An empty list is returned if no directory is found, or if the directory in argument does not exist. The function does not look inside sub-directories. The order of the directories in the output list is undefined (can be anything): if you need a sorted list, you have to sort it yourself. Lastly, the path given in argument can contain wildcard characters *, like in bash, to filter out the output list.

Example

```
vec1s d = file::list_directories("/path/to/phypp/");  
d; // {"bin", "cmake", "doc", "include", "test", "tools"}  
  
d = file::list_directories("/path/to/phypp/t*");  
d; // {"test", "tools"}
```

- **vec1s** file::list_files(string)

This function scans the directory given in argument and returns the list of all files it contains (excluding directories). An empty list is returned if no file is found, or if the directory in argument does not exist. The function does not look inside sub-directories. The order of the files in the output list is undefined (can be anything): if you need a sorted list, you have to sort it yourself. Lastly, the path given in argument can contain wildcard characters `*`, like in `bash`, to filter out the output list.

Example

```
vec1s d = file::list_files("/path/to/phypp/doc");  
d; // {"compile.sh", "phypp.pdf", "phypp.tex"}  
  
d = file::list_files("/path/to/phypp/doc/*.tex");  
d; // {"phypp.tex"}
```

- \textcircled{V} `string file::directorize(string)`

This function modifies the path given in argument to make sure that it can be used as a directory, i.e. in UNIX systems it makes sure that it ends with a forward slash `/`. The goal is to produce a directory path that can be appended the base name of a file to form a valid file path.

Example


```
std::string p = file::directorize("/some/path");  
p; // "/some/path/"  
  
p = file::directorize("/another/path");  
p; // "/another/path/"
```

- \textcircled{V} `string file::get_basename(string)`

This function extracts the name of a file from its full path given in argument. If this path is that of a directory, the function returns the name of this directory. This behavior is similar to the `bash` function `basename`.

Example


```
std::string n = file::get_basename("/some/path");  
n; // "path"  
  
n = file::get_basename("/another/path/to/a/file.txt");  
n; // "file.txt"
```

-  `string remove_extension(string)`

This function scans the provided string to look for a file extension (e.g., ".fits"). The extension is whatever ends the string after the *last* dot (and including this dot). If one is found, the function will return a new string with the extension removed. Else, the input string is returned untouched.

Example


```
vec1s v = {"p1_m2.txt", "p3_c4.fits", "p1_t8.dat.fits", "readme"};  
vec1s s = remove_extension(v);  
s; // {"p1_m2", "p3_c4", "p1_t8.dat", "readme"}
```

-  `string file::get_directory(string)`

This function extracts the path of the directory that contains the file given in argument. If the path in argument is that of a directory, the function returns the path of its parent directory. This behavior is similar to the bash function `dirname`, except that the returned path ends with a forward slash `/`.

Example

```
std::string n = file::get_directory("/some/path");  
n; // "/some/"  
  
n = file::get_directory("/another/path/to/a/file.txt");  
n; // "/another/path/to/a/"
```

-  `bool file::mkdir(string)`

This function creates a new directory whose path is given in argument and returns **true**. If the directory could not be created (e.g., because of permission issues), the function returns **false**. If the directory already exists, the function does nothing and returns **true**.

Example

```
bool b = file::mkdir("/path/to/phypp/a/new/directory");  
// Will most likely create the directories:  
// - /path/to/phypp/a  
// - /path/to/phypp/a/new  
// - /path/to/phypp/a/new/directory  
b; // maybe true or false, depending on your permissions
```

- **bool** file::copy(string from, to)

This function creates a copy of the file `from` at the location given in `to` and returns **true**. If the new file could not be created (e.g., because of permission issues), or if the file `from` could not be found or read, the function returns **false**. If the file `to` already exists, it will be overwritten.

Example

```
bool b = file::copy("/home/joe/.phypprc", "/home/bob/.phypprc");  
b; // maybe true or false, depending on your permissions
```

- **bool** file::remove(string)

This function will delete the file given in argument and return **true** on success, or if the file does not exist. It will return **false** only if the file exists but could not be removed (i.e., because you are lacking the right permissions).

Example

```
// That's a bad idea, but for the sake of the example...  
bool b = file::remove("/home/joe/.phypprc");  
b; // probably true
```

- `string file::to_string(string)`

This function reads the content of the file whose path is given in argument, stores all the characters (including line break characters and spaces) inside a string and returns it. If the file does not exist, the function returns an empty string.

Example

```
std::string r = file::to_string("/etc/lsb-release");
// 'r' now contains all the lines of the file, each
// separated by a newline character '\n'.
```

3.5.2 ASCII table input/output

- `void file::read_table(string f, uint_t s, ...)`

This function reads the content of the ASCII table `f` and stores the data inside vectors. The second argument `s` gives the number of lines to skip before starting reading data: this is useful to ignore the lines that correspond to the header of the table. Note that, due to the lack of standard formatting for the header, there is no automatic way to ignore these lines.

The vectors in which the data should be stored are given in argument, one after the other. Each column of the file will be stored in a different vector, in the order in which they are provided to the function. If there is more columns than vectors, the extra columns are ignored. If there is more vectors than columns, the program will stop and report an error.

Columns in the file can be separated by any number of spaces and tabulations. While this is better for human readability, columns need not be aligned to be read by this function. However, “holes” in the table are not supported, i.e., rows that only have data for some columns but not all. If such a situation is encountered, the “hole” is considered as white spaces and ignored, and the data from this column is actually read from the next one. This will make the function report an error only if all columns of the file are read. For this reason, it is advised to always read all the columns of a given file, even those that are not used (see below for an efficient way to do that). The table may contain empty lines however, they will simply be ignored.

Regarding the type of the data. ASCII tables are not strongly typed: as long as the data in the table can be converted to a value of its corresponding vector, this function will be able to read it. There are particular rules though:

- Strings may not contain spaces (since they will be understood as column separators). Adding quotes `"..."` will *not* change that.
- Floating point columns can contain special values such as NaN or the infinity. These can be spelled `nan`, `+nan`, `-nan`, `+inf`, `-inf`, `inf`, `inf+` or `inf-` (case does not matter). The special value `null` is also accepted and means zero. Both fixed point and scientific notation is allowed.
- In general for arithmetic data, if the value to be read is too large to fit in the corresponding C++ variable, the program will stop report the error. This will happen for example is trying to read a number like `1e128` inside a `float` vector. Use a larger data type to fix this (e.g. `double` in this particular case).

In the following examples, we will parse a table which contains the following columns:

```
# - source ID          (integer)
# - right ascension    (double)
# - declination        (double)
# - badpixel flag      (integer)
# - contamination flag (integer)
# - cov 24             (float)
# - cov 160            (float)
# - flux 24            (float)
# - error 24           (float)
# - flux 160           (float)
# - error 160          (float)
```

This is the header of the table, and it contains 11 lines that should be ignored.

Here is how to read the first three columns:

```
// First declare the vectors and their types
vec1u id;
vec1d ra, dec;

// Then read the data
file::read_table("some/ascii_table.dat", 11,
    id, ra, dec // read three columns
);
```

The placeholder character `_` can be used in the argument list to skip a given column. Let's say we want to read the contamination flag, but we don't care about the badpixel flag. We can write:

```

// First declare the vectors and their types
vec1u id;
vec1d ra, dec;
vec1u contam;

// Then read the data
file::read_table("some/ascii_table.dat", 11,
    id, ra, dec, // read three columns
    _, contam    // ignore one column, and read one column
);

```

Lastly, the `file::columns(n, ...)` function can be used to group several columns and repeat `n` times the same extraction pattern. This can be used to read multiple columns inside a single two-dimensional vector. Let's say we want now to read the data from `cov24` and `cov160` in a single vector:

```

// First declare the vectors and their types
vec1u id;
vec1d ra, dec;
vec1u contam;
vec2f cov;

// Then read the data
file::read_table("some/ascii_table.dat", 11,
    id, ra, dec, // read three columns
    _, contam,   // ignore one column, and read one column
    file::columns(2, cov) // read two columns in one vector
);
cov(_,0); // contains data from "cov 24"
cov(_,1); // contains data from "cov 160"

```

The `file::columns(n, ...)` function can also be used to ignore multiple columns at a time. Let's say we don't want to read the contamination flag anymore, so we need to skip two consecutive columns. We could write `_, _`, but this becomes tedious and error prone when many columns are to be ignored. Instead, we can write:

```

// First declare the vectors and their types
vec1u id;
vec1d ra, dec;
vec2f cov;

```

```
// Then read the data
file::read_table("some/ascii_table.dat", 11,
  id, ra, dec,          // read three columns
  file::columns(2, _),  // ignore two columns
  file::columns(2, cov) // read two columns in one vector
);
```

The `file::columns(n, ...)` function can also be used to read consecutive groups of columns with a given pattern. Let's say we now want to read the flux and error inside their respective two-dimensional vectors. The pattern here is "flux, error, flux, error, ...". We can parse this using:

```
// First declare the vectors and their types
vec1u id;
vec1d ra, dec;
vec2f cov, flux, err;

// Then read the data
file::read_table("some/ascii_table.dat", 11,
  id, ra, dec,          // read three columns
  file::columns(2, _),  // ignore two columns
  file::columns(2, cov), // read two columns in one vector
  file::columns(2, flux, err) // read (flux,err,flux,err)
);
```

Finally, note that it is possible to mix together vectors and `_` in this pattern. Imagine we just want to read the flux columns and ignore the errors. We can write:

```
// First declare the vectors and their types
vec1u id;
vec1d ra, dec;
vec2f cov, flux;

// Then read the data
file::read_table("some/ascii_table.dat", 11,
  id, ra, dec,          // read three columns
  file::columns(2, _),  // ignore two columns
  file::columns(2, cov), // read two columns in one vector
  file::columns(2, flux, _) // read (flux,_,flux,_)
);
```

- **void** file::write_table(string f, uint_t w, ...)
- void** file::write_table_csv(string f, uint_t w, ...)
- void** file::write_table_hdr(string f, uint_t w, ...)

This collection of functions will write the data contained inside one of multiple vectors into a file *f*, in human-readable “ASCII” format¹. The data is written in separate columns, with a fixed width of *w* characters. Spaces are used to fill the empty space between columns.

The first two functions in the list will just write all the data, and nothing else. The difference between the two is that the second version uses the “CSV” format (comma-separated values), i.e., appends a comma after each value. Apart from this, the functions behave the same: after the column width *w*, the following arguments can be any number of 1D vectors. They must contain the same number of elements. 2D vectors are also allowed, in which case, by convention, the first dimension is considered as the “row” dimension, and the second is the “column” dimension.

Note that this function will only create the *file*. If the path *f* given in argument contains directories that do not exist, the function will fail. You have to create them yourself beforehand.

For example, imagine we want to save the following data into a file:

```
// Some arbitrary data
vec1u id = {1,2,3,4,5};
vec1i x = {125,568,9852,12,-51};
vec1i y = {-56,157,2,99,1024};
vec2f flux(5,2);
flux(_,0) = {0.0, 1.2, 5.6, 9.5, 1.5};
flux(_,1) = {9.6, 0.0, 4.5, 0.0, 0.0};
```

We can use this function to create an ASCII table with columns that are 8 characters wide:

```
// Write these in a file
file::write_table("some/path/to/a_table.dat", 8,
    id, x, y, flux
);
```

¹This is simple and convenient for small files, but if the volume of data is huge, consider instead using binary FITS files (3.10.4). This will be both faster to read and write, and will also occupy less space on your disk.

The content of the file will be:

1	125	-56	0	9.6
2	568	157	1.2	0
3	9852	2	5.6	4.5
4	12	99	9.5	0
5	-51	1024	1.5	0

Sometimes it is desirable to also add some information in the file about what kind of data is stored in each column. This is usually done with a *header*, i.e., a couple of lines at the beginning of the file containing information about what the file contains, among other things. This can be done easily using the third function in the list above, `file::write_table_hdr()`. Usage is fairly simple:

```
// Write these in a file
file::write_table_hdr("some/path/to/a_table.dat", 8,
    {"id", "x", "y", "flux_1", "flux_2"},
    id, x, y, flux
);
```

The resulting file is:

#	id	x	y	flux_1	flux_2
#					
	1	125	-56	0	9.6
	2	568	157	1.2	0
	3	9852	2	5.6	4.5
	4	12	99	9.5	0
	5	-51	1024	1.5	0

However it can be tedious to write the name of the columns manually, especially when they match closely the names of the vectors in the C++ code. For this reason, another way to write the above file is to use an alternative version of this function and the `ftable()` macro:

```
// Write these in a file
file::write_table_hdr("some/path/to/a_table.dat", 8,
    ftable(id, x, y, flux)
);
```

This will take the C++ name of the variables and will use them directly as column names. For 2D vectors, “_i” is appended to the name of the vector for each column

i. If you need better looking headers, you will have to write them manually using the previous signature.

Finally, if you want to fine tune the way a particular column is written in the file, for example if you want to use scientific notation, you can perform the conversion to string yourself (3.3.1), and feed string vectors to this function.

3.6 Printing to the terminal

- **void** print(...)
void error(...)
void warning(...)
void note(...)
- **bool** prompt(string msg, T& v, string err = "")

3.7 Measuring time

- **double** now()
- string today()
- string time_str(**double**)
- string seconds_str(**double**)
- **auto** progress_start(uint_t)
void progress(**auto** p, uint_t m)
void print_progress(**auto** p, T i, uint_t m)

3.8 Mathematics

The mathematics support library is among the largest inside *phy++*. It contains many functions, as well as a handful of useful constants. Some functionalities of this library are only available if you have installed the *fftw* and *gsl* libraries. If not, the specific functions that depend on these libraries will not be available, or will be slow, but the rest of the library will function properly.

This library provides the following global constants:

- `fnan` and `dnan`. These are the **float** and **double** representation of the “not-a-number” (NaN) special value. This value is returned by some operations that are mathematically undefined in the real domain. For example, dividing zero by zero, or taking the square root of a negative number. NaN has some very peculiar properties that can surprise the newcomer. In particular, it propagates extremely fast, since any operation involving at least a NaN value will always return NaN, e.g., `2.0 + fnan == fnan`). More troubling, any comparison operation involving a NaN will return **false**, e.g., `(10.0 < fnan) == false` and `(10.0 >= fnan) == false` too. The only notable exception to this rule is that `(fnan != fnan) == true`. Knowing this, NaN is a very useful return value to indicate that giving an actual value would not make sense. For example, in a galaxy catalog, some galaxies may have been observed at a certain wavelength, but not all of them. For those that are not observed, we do not know their flux. In this case, astronomers typically assign them a special, weird value, such as -99. Using NaN in this case is clearer.
- `fpi` and `dpi`. This is the **float** and **double** closest representation of the number $\pi = 3.14519\dots$
- `finf` and `dinf`. This is the **float** and **double** representation of the positive infinity. The positive infinity is larger than any other finite value.

We now present the functions provided by this support library. One of the responsibilities of this library is to bring vectorized versions of standard mathematical functions that only work for scalar values. Since these functions are fairly common and well known, we will not describe their signature and behavior, and instead just list them here:

- exponentiation: `sqrt`, `pow`,
- trigonometry: `cos`, `sin`, `tan`, `acos`, `asin`, `atan`, `cosh`, `sinh`, `tanh`, `acosh`, `asinh`, `atanh`,
- exponentials and logarithms: `exp`, `log`, `log2`, `log10`,
- special functions: `erf`, `erfc`, `tgamma`,
- rounding: `ceil`, `floor`, `round`,
- absolute value: `fabs`.

We also introduce the functions `bessel_j0`, `bessel_j1`, `bessel_y0`, `bessel_y1`, `bessel_i0`, `bessel_i1`, `bessel_k0`, `bessel_k1`. The scalar version of the first four are provided by the C++ standard, while the last four are provided by the `gsl`.

We now list the other, less common functions provided in this library. These are grouped by sections.

3.8.1 Low level mathematics

- \textcircled{V} **double** e10(**double**)
- \textcircled{V} T sqr(T)
- \textcircled{V} T invsqr(T)
- \textcircled{V} T clamp(T t, mi, ma)
- \textcircled{V} **bool** is_finite(T)
- \textcircled{V} **bool** is_nan(T)
- \textcircled{V} int_t sign(T)

3.8.2 Sequences and bins

- vec1u rgen(T)
vec<1,T> rgen(T i, j)
vec1d rgen(T i, j, n)
vec1d rgen_log(T i, j, n)
- vec<2,T> make_bins(T mi, ma)
vec<2,T> make_bins(T mi, ma, uint_t n)
vec<2,T> make_bins(vec<1,T>)
- \textcircled{V} **bool** in_bin(T v, vec<1,U> b)
 \textcircled{V} **bool** in_bin(T v, vec<2,U> b, uint_t ib)
 \textcircled{V} **bool** in_bin_open(T v, vec<2,U> b, uint_t ib)
- vec<1,T> bin_center(vec<2,T>) T bin_center(vec<1,T>)
- vec<1,T> bin_width(vec<2,T>) T bin_width(vec<1,T>)

3.8.3 Randomization

- **auto** make_seed(T)
- **double** randomn(**auto** seed)
vec<N,**double**> randomn(**auto** seed, ...)
- **double** randomu(**auto** seed)
vec<N,**double**> randomu(**auto** seed, ...)
- T randomi(**auto** seed, T mi, T ma)
vec<N,T> randomi(**auto** seed, T mi, T ma, ...)
- T random_pdf(**auto** seed, vec<1,T> x, vec<1,U> y, ...)
- **bool** random_coin(**auto** seed, **double** p)
vec<N,**bool**> random_coin(**auto** seed, **double** p, ...)
- vec shuffle(vec, **auto** seed)
void inplace_shuffle(vec&, **auto** seed)

3.8.4 Reduction

- **void** run_dim_idx(uint_t d, F f, ...)
- vec<D-1,T> run_dim(vec<D,T> v, uint_t d, F f)
- T total(vec<D,T> v)
vec<D-1,T> partial_total(uint_t d, vec<D,T> v)
- uint_t count(vec<D,**bool**> v)
vec<D-1,uint_t> partial_count(uint_t d, vec<D,**bool**> v)
- uint_t fraction_of(vec<D,**bool**> v)
vec<D-1,uint_t> partial_fraction_of(uint_t d, vec<D,**bool**> v)
- **double** mean(vec v)
vec<D-1,**double**> partial_mean(uint_t d, vec<D,T> v)

- `T median(vec<D,T> v)`
`T inplace_median(vec<D,T>& v)`
`vec<D-1,T> partial_median(uint_t d, vec<D,T> v)`
- `T percentile(vec<D,T> v, double p)`
`vec<D-1,T> partial_percentile(uint_t d, vec<D,T> v, double p)`
`vec<1,T> percentiles(vec<D,T> v, double ...)`
- `T min(vec<D,T> v)`
`T min(vec<D,T> v, uint_t& i)`
`vec<D-1,T> partial_min(uint_t d, vec<D,T> v)`
`T max(vec<D,T> v)`
`T max(vec<D,T> v, uint_t& i)`
`vec<D-1,T> partial_max(uint_t d, vec<D,T> v)`
- `uint_t min_id(vec v)`
`uint_t max_id(vec v)`
- `vec1u histogram(vec v, vec<2,V> b)`
`vec<1,W> histogram(vec<D,T> v, vec<D,W> w, vec<2,U> b)`
- `vec2u histogram2d(vec x, y, vec<2,U> bx, by)`
`vec2u histogram2d(vec x, y, vec<2,U> bx, by, F func)`
- `vec<D,bool> sigma_clip(vec<D,T> v, double pl, double pu)`
- `vec<D,bool> mad_clip(vec<D,T> v, double s)`

3.8.5 Interpolation

- `double interpolate(double y1, y2, x1, x2, x)`
`T interpolate(vec<D,T> y, vec<D,U> x, V nx)`
`vec<D2,T> interpolate(vec<D1,T> y, vec<D1,U> x, vec<D2,V> nx)`
- `T bilinear(vec<2,T> m, double x, y)`
- `vec<2,T> rebin(vec<2,T> m, vec1d mx, my, nx, ny)`
- `void merge_add(vec<1,T> x1, x2, y1, y2, x, y)`

3.8.6 Calculus

- `double` derivatel(F f, `double` x, e)
`double` derivatel(F f, vec1d x, `double` e, uint_t i)
- `double` derivate2(F f, `double` x, e)
`double` derivate2(F f, vec1d x, `double` e, uint_t i)
- `double` integrate_trap(F f, `double` x, uint_t n)
`double` integrate(F f, `double` x, e = `default`)
- `double` integrate(vec<1,T> x, y)
`double` integrate(vec<1,T> x, y, `double` x0, x1)
`double` integrate(vec<2,T> x, y)
`double` integrate(vec<2,T> x, y, `double` x0, x1)

3.8.7 Algebra

- `vec<2,T> mmul(vec<2,T> a, b)`
`vec<1,V> mmul(vec<2,T> a, vec<1,U> b)`
`vec<1,V> mmul(vec<1,T> b, vec<2,U> a)`
- `vec<2,T> transpose(vec<2,T>)`
- `vec<1,T*> diagonal(vec<2,T>)`
- `vec2d identity_matrix(uint_t)`
- `vec2d scale_matrix(double sx, sy)`
`vec2d scale_matrix(double s)`
- `vec2d translation_matrix(double dx, dy)`
- `vec2d rotation_matrix(double a)`
- `vec1d point2d(double x, y)`
- `[LAPACK] bool invert(vec2d a, vec2d& i)`
`[LAPACK] bool inplace_invert(vec2d&)`

- [LAPACK] **bool** invert_symmetric(vec2d a, vec2d& i)
[LAPACK] **bool** inplace_invert_symmetric(vec2d&)
- [LAPACK] **bool** solve_symmetric(vec2d a, vec1d b, vec1d& r)
[LAPACK] **bool** inplace_solve_symmetric(vec2d& a, vec1d& b)
- [LAPACK] **bool** eigen_symmetric(vec2d a, vec1d& va, vec1d& ve)
[LAPACK] **bool** inplace_eigen_symmetric(vec2d& a, vec1d& va)
- [FFTW] vec2cd fft(vec2d)
[FFTW] vec2d ifft(vec2cd)
- vec<1,W> convolve(vec<1,T> x, vec<1,U> y, vec<1,V> k)

3.8.8 Fitting

- **auto** linfit(vec y, e, ...)
auto linfit_pack(vec y, e, x)
- **auto** affinefit(vec y, e, x)
- **auto** mpfit(F d, vec1d p, **auto** opt = **default**)
- **auto** mpfitfun(vec y, e, x, F f, vec1d p, **auto** opt = **default**)

3.8.9 Geometry

- vec1u convex_hull(vec x, y)
- **bool** is_hull_closed(vec1u h, vec hx, hy)
- \mathcal{V} **bool** in_convex_hull(T x, y, vec1u h, vec hx, hy)
- \mathcal{V} **double** convex_hull_distance(T x, y, vec1u h, vec hx, hy)
- \mathcal{V} **double** angdistr(**double** ra1, dec1, ra2, dec2)
 \mathcal{V} **double** angdist(**double** ra1, dec1, ra2, dec2)
- vec<D,**bool**> angdist_less(vec<D,**double**> ra1, dec1, **double** ra2, dec2)
- **void** move_ra_dec(**double**& ra, dec, **double** dra, ddec)
void move_ra_dec(vec<D,**double**>& ra, dec, **double** dra, ddec)

3.8.10 Debug functions

- `void data_info(vec)`
- `void mprint(vec<2,T>)`

3.9 Parallel execution

- `void fork(string)`
`void spawn(string)`
- `auto thread::pool(uint_t)`
- `void thread::sleep_for(double)`

3.10 FITS input/output

The FITS (Flexible Image Transport System) format is a general purpose file format developed for astrophysics data. In particular, FITS files can store images with floating point pixel values, image cubes, but also binary data tables with an arbitrary number of columns and rows. Using a meta-data system (FITS keywords), FITS files usually carry a number of important additional informations about their content. E.g., for images files, the mapping between image pixels and sky coordinates (WCS coordinates), or the physical unit of the pixel values.

Storing data tables in binary inside FITS files is a space-efficient and fast way to store and read non-image data. FITS tables come in two fashions: row-oriented and column-oriented tables. In row-oriented tables, all the data about one row (e.g., about one galaxy in the table) is stored contiguously on disk. This means that it is very fast to retrieve all the information about a given object. In column-oriented tables however, a whole column is stored contiguously in memory. This means that it is very fast to read a given column for all the objects in the table. Due to the way the *phy++* library is designed, it is more efficient to use the latter format, since a given column of the file will be represented by a single *phy++* vector. It also brings the nice advantage of allowing to store columns of different lengths, e.g. to combine two tables in the same file, or to carry meta-data that would be hard to store in the standard FITS keywords. The column-oriented format is not well known, but most softwares and libraries do support it².

²Topcat does. In IDL, column-oriented FITS files are supported by the `mrdfits` and `mwrfits` procedures.

Finally, note that this support library introduces a new type: `fits::header` (that we will shorten to `header` in the following). This type is used to store the header of any FITS file. For now, it is actually just a raw `std::string`, but that might change in the future.

We now describe the various functions offered by this library, split into categories.

3.10.1 Generic header functions

- `header fits::read_header(string f)`
`header fits::read_header_hdu(string f, uint_t hdu)`
`header fits::read_header_sectfits(string f, uint_t s)`
- **bool** `fits::getkey(header hdr, string k, T& v)`
- **bool** `fits::setkey(header hdr, string k, T& v, string c = "")`

3.10.2 FITS images input/output

- `uint_t fits::file_axes(string)`
- `vec1u fits::file_dimensions(string)`
- **bool** `fits::is_cube(string)`
- **bool** `fits::is_image(string)`
- **void** `fits::read(string f, vec& v)`
void `fits::read(string f, vec& v, header& hdr)`
void `fits::read_hdu(string f, vec& v, uint_t hdu)`
void `fits::read_hdu(string f, vec& v, uint_t hdu, header& hdr)`
- `vec1s fits::read_sectfits(string f)`
- **void** `fits::write(string f, vec v)`
void `fits::write(string f, vec v, header hdr)`
- **void** `fits::display(string f)`
void `fits::display(string r, string g)`
void `fits::display(string r, string g, string b)`

3.10.3 WCS coordinates

- **bool** fits::make_wcs_header(**auto** p, header& hdr)
bool fits::make_wcs_header(vec1s p, header& hdr)
- header fits::filter_wcs(header)
- **auto** fits::extast(header)
- **void** fits::ad2xy(**auto** wcs, vec ra, dec, vec& x, y)
void fits::xy2ad(**auto** wcs, vec ra, dec, vec& x, y)
- **bool** fits::get_pixel_size(string file, **double**& a)

3.10.4 FITS tables input/output

- vec<1,**auto**> fits::read_table_columns(string)
- **void** fits::read_table(string f, ...)
void fits::read_table_loose(string f, ...)
- **void** fits::write_table(string f, ...)
void fits::update_table(string f, ...)

3.11 Image processing

- vec<2,T> enlarge(vec<2,T> v, array {l1, l2, u1, u2}, T d = 0)
vec<2,T> enlarge(vec<2,T> v, uint_t u, T d = 0)
- vec<2,T> shrink(vec<2,T> v, array {l1, l2, u1, u2})
vec<2,T> shrink(vec<2,T> v, uint_t u)
- **void** subregion(vec<2,T> v, vec1i r, vec1u& rr, rs)
void subregion(vec<2,T> v, vec1i r, T d = 0)
- vec<2,T> translate(vec<2,T> v, **double** x, y, T d = 0)
- vec<2,T> flip_x(vec<2,T> v)

- `vec<2,T> flip_y(vec<2,T> v)`
- `vec<2,T> scale(vec<2,T> v, double s)`
- `vec<2,T> rotate(vec<2,T> v, double a)`
- `vec2d circular_mask(vec1d d, vec1d c, double r)`
- `vec<1,T> radial_profile(vec<2,T> m, uint_t n)`
- `vec<2,T> generate_img(array {w,h}, F f)`
- `vec2d gaussian_profile(array {w,h}, double sigma)`
- `vec<2,T> convolve2d(vec<2,T> m, vec<2,U> k)`
- `vec<2,T> boxcar(vec<2,T> m, uint_t n, F f)`
- `vec2b mask_inflate(vec2b m, uint_t d)`

3.12 Astrophysics

3.12.1 PSF fitting

- `bool make_psf(array<uint_t> {w, h}, double x0, y0, string model, vec2d& psf)`
- `auto psffit(vec<2,T> m, e, psf, vec1i pos)`
`auto psffit(vec<2,T> m, e, psf)`

3.12.2 Cosmology

- `auto get_cosmo()`
`vec1s cosmo_list()`
`auto cosmo_wmap()`
`auto cosmo_plank()`
`auto cosmo_std()`
- \mathcal{V} `T lumdist(T z, auto cosmo)`

- \textcircled{V} T lookback_time(T z, **auto** cosmo)
- \textcircled{V} T vuniverse(T z, **auto** cosmo)
- \textcircled{V} T propsize (T z, **auto** cosmo)

3.12.3 Fluxes, magnitudes and luminosities

- T lsun2uJy(T z, d, lam, lum)
T uJy2lsun(T z, d, lam, lum)
- T lsun2mag(T lam, lum, **double** zp = 23.9)
T mag2lsun(T lam, lum, **double** zp = 23.9)
- T uJy2mag(T flx, **double** zp = 23.9)
T mag2uJy(T mag, **double** zp = 23.9)
- **auto** read_filter_db(string)
- **auto** get_filter(**auto** db, string name)
auto get_filters(**auto** db, vec1s names)
- **double** sed2flux(**auto** fil, vec<1,T> lam, sed)
vec1d sed2flux(**auto** fil, vec<2,T> lam, sed)
- **double** sed_convert(**auto** from, to, **double** z, d, vec<1,T> lam, sed)
- **double** lir_8_1000(vec<1,T> lam, sed)

3.12.4 Sky coordinates

- **double** field_area_hull(vec1u hull, vec ra, dec)
double field_area_hull(vec ra, dec)
- **double** field_area_h2d(vec ra, dec)
- **double** field_area(vec ra, dec)
- vec1d angcorrel(vec<D1,T> ra, dec, vec<D2,U> rra, rdec, vec<2,V> b)

- `auto randpos_uniform(auto seed, vec1d rra, rdec, F in, vec& ra, dec, auto options = default)`
- `auto randpos_power_circle(auto seed, double ra0, dec0, r0, vec& ra, dec, auto options = default)`
- `auto randpos_power(auto seed, vec1u h, vec<D1,double> hra, hdec, vec& ra, dec, auto options = default)`
- \textcircled{V} `bool sex2deg(string sra, sdec, T& ra, dec)`
- \textcircled{V} `void deg2sex(T ra, dec, string& sra, sdec)`
- `auto qxmatch(vec<1,T> ra1, dec1, ra2, dec2, auto options = default)`
- `vec2d qdist(vec<1,T> ra, dec, auto options = default)`

3.12.5 Catalog management

- `bool get_band(auto cat, string band, uint_t& b)`
- `bool get_band(auto cat, string band, note, uint_t& b)`

3.12.6 Image stacking

- `bool pick_sources(vec<2,T> m, vec1d x, y, uint_t hs, vec<3,T>& c, vec1u& ids)`
- `auto qstack(vec<1,T> ra, dec, string ff, uint_t hs, vec<3,T>& fc, vec1u& i, auto options = default)`
- `auto qstack(vec<1,T> ra, dec, string ff, fw, uint_t hs, vec<3,T>& fc, wc, vec1u& i, auto options = default)`
- `auto qstack_mean(vec<3,T> fc)`
- `auto qstack_mean(vec<3,T> fc, wc)`
- `auto qstack_median(vec<3,T> fc)`

- `void qstack_bootstrap(vec<3,T> fc, uint_t nb, ns, auto seed, F f)`
`void qstack_bootstrap(vec<3,T> fc, wc, uint_t nb, ns, auto seed, F f)`
`void qstack_bootstrap(uint_t nb, ns, auto seed, F f, ...)`
- `vec<3,T> qstack_mean_bootstrap(vec<3,T> fc, uint_t nb, ns, auto seed)`
`vec<3,T> qstack_mean_bootstrap(vec<3,T> fc, wc, uint_t nb, ns, auto seed)`
- `vec<3,T> qstack_median_bootstrap(vec<3,T> fc, uint_t nb, ns, auto seed)`

3.12.7 Template fitting

- `vec2d template_observed(auto lib, double z, d, auto filters)`
`vec2d template_observed(auto lib, vec1d z, d, auto filters)`
- \textcircled{V} `double limweight(double)`
- `auto template_fit(auto lib, auto seed, T z, d, vec1d flux, err,`
`auto filters, auto options = default)`

Chapter 4

Tools

4.1 Astrophysics

4.1.1 `angcorrel`

4.1.2 `catinfo`

4.1.3 `deg2sex` and `sex2deg`

4.1.4 `findsrc`

4.1.5 `fluxcube`

4.1.6 `getgal`

4.1.7 `photinfo`

4.1.8 `psffit`

4.1.9 `qstack2`

4.1.10 `qxmacth2`

4.1.11 `randsrc`

4.1.12 `subsrc`

4.2 FITS and ASCII

4.2.1 `fits2ascii`

92

4.2.2 `fitstool`

4.2.3 `imgtool`

4.2.4 `qconvol`

4.2.5 `remcol`

Index

acos, 78
acosh, 78
affinefit, 83
align_center, 64
align_left, 64
align_right, 64
angcorrel, 88, 92
angdist, 83
angdist_less, 83
angdistr, 83
append, 45
asin, 78
asinh, 78
astar_find, 42
atan, 78
atanh, 78

bessel_i0, 78
bessel_i1, 78
bessel_j0, 78
bessel_j1, 78
bessel_k0, 78
bessel_k1, 78
bessel_y0, 78
bessel_y1, 78
bilinear, 81
bin_center, 79
bin_width, 79
bounds, 39

boxcar, 87

catinfo, 92
ceil, 78
circular_mask, 87
clamp, 79
collapse, 58
complement, 39
convex_hull, 83
convex_hull_distance, 83
convolve, 83
convolve2d, 87
cos, 78
cosh, 78
cosmo_list, 87
cosmo_plank, 87
cosmo_std, 87
cosmo_wmap, 87
count, 80
cut, 56

data_info, 84
deg2sex, 89, 92
derivate1, 82
derivate2, 82
diagonal, 82
dindgen, 36
dinf, 78
distance, 65
dnan, 78

- dpi, [78](#)
- e10, [79](#)
- eigen_symmetric, [83](#)
- empty, [54](#)
- end_with, [62](#)
- enlarge, [86](#)
- equal_range, [40](#)
- erase_begin, [63](#)
- erase_end, [63](#)
- erf, [78](#)
- erfc, [78](#)
- error, [77](#)
- exp, [78](#)
- fabs, [78](#)
- fft, [83](#)
- field_area, [88](#)
- field_area_h2d, [88](#)
- field_area_hull, [88](#)
- file::copy, [70](#)
- file::directorize, [68](#)
- file::exists, [66](#)
- file::get_basename, [68](#)
- file::get_directory, [69](#)
- file::is_older, [67](#)
- file::list_directories, [67](#)
- file::list_files, [67](#)
- file::mkdir, [69](#)
- file::read_table, [71](#)
- file::remove, [70](#)
- file::to_string, [71](#)
- file::write_table, [75](#)
- file::write_table_csv, [75](#)
- file::write_table_hdr, [75](#)
- find, [58](#)
- findgen, [36](#)
- findsrc, [92](#)
- finf, [78](#)
- fits2ascii, [92](#)
- fits::ad2xy, [86](#)
- fits::display, [85](#)
- fits::extast, [86](#)
- fits::file_axes, [85](#)
- fits::file_dimensions, [85](#)
- fits::filter_wcs, [86](#)
- fits::get_pixel_size, [86](#)
- fits::getkey, [85](#)
- fits::is_cube, [85](#)
- fits::is_image, [85](#)
- fits::make_wcs_header, [86](#)
- fits::read, [85](#)
- fits::read_hdu, [85](#)
- fits::read_header, [85](#)
- fits::read_sectfits, [85](#)
- fits::read_table, [86](#)
- fits::read_table_columns, [86](#)
- fits::read_table_loose, [86](#)
- fits::setkey, [85](#)
- fits::update_table, [86](#)
- fits::write, [85](#)
- fits::write_table, [86](#)
- fits::xy2ad, [86](#)
- fitstool, [92](#)
- flat_id, [36](#)
- flatten, [43](#)
- flip_x, [86](#)
- flip_y, [87](#)
- floor, [78](#)
- fluxcube, [92](#)
- fnan, [78](#)
- fork, [84](#)
- fpi, [78](#)
- fraction_of, [80](#)
- from_string, [52](#)
- gaussian_profile, [87](#)
- generate_img, [87](#)

get_band, 89
 get_cosmo, 87
 get_filter, 88
 get_filters, 88
 getgal, 92

 hash, 53
 histogram, 81
 histogram2d, 81

 identity_matrix, 82
 ifft, 83
 imgtool, 92
 in_bin, 79
 in_bin_open, 79
 in_convex_hull, 83
 indgen, 36
 inplace_eigen_symmetric, 83
 inplace_invert, 82
 inplace_invert_symmetric, 83
 inplace_median, 81
 inplace_remove, 45
 inplace_shuffle, 80
 inplace_solve_symmetric, 83
 inplace_sort, 37
 integrate, 82
 integrate_trap, 82
 interpolate, 81
 invert, 82
 invert_symmetric, 83
 invsqr, 79
 is_any_of, 41
 is_finite, 79
 is_hull_closed, 83
 is_nan, 79
 is_sorted, 37

 keep_end, 64
 keep_start, 64

 length, 54
 limweight, 90
 linfit, 83
 linfit_pack, 83
 lir_8_1000, 88
 log, 78
 log10, 78
 log2, 78
 lookback_time, 88
 lower_bound, 39
 lsun2mag, 88
 lsun2uJy, 88
 lumdist, 87

 mad_clip, 81
 mag2lsun, 88
 mag2uJy, 88
 make_bins, 79
 make_psf, 87
 make_seed, 80
 mask_inflate, 87
 match, 41
 match_dictionary, 42
 max, 81
 max_id, 81
 mean, 80
 median, 81
 merge_add, 81
 min, 81
 min_id, 81
 mmul, 82
 move_ra_dec, 83
 mpfit, 83
 mpfitfun, 83
 mprint, 84
 mult_ids, 35

 note, 77
 now, 77

- [npos](#), [34](#)
- [partial_count](#), [80](#)
- [partial_fraction_of](#), [80](#)
- [partial_max](#), [81](#)
- [partial_mean](#), [80](#)
- [partial_median](#), [81](#)
- [partial_min](#), [81](#)
- [partial_percentile](#), [81](#)
- [partial_total](#), [80](#)
- [percentile](#), [81](#)
- [percentiles](#), [81](#)
- [photinfo](#), [92](#)
- [phypp_check](#), [46](#)
- [pick_sources](#), [89](#)
- [point2d](#), [82](#)
- [pow](#), [78](#)
- [prepend](#), [45](#)
- [print](#), [77](#)
- [print_progress](#), [77](#)
- [progress](#), [77](#)
- [progress_start](#), [77](#)
- [prompt](#), [77](#)
- [proptype](#), [88](#)
- [psffit](#), [87](#), [92](#)
- [qconvol](#), [92](#)
- [qdist](#), [89](#)
- [qstack](#), [89](#)
- [qstack2](#), [92](#)
- [qstack_bootstrap](#), [90](#)
- [qstack_mean](#), [89](#)
- [qstack_mean_bootstrap](#), [90](#)
- [qstack_median](#), [89](#)
- [qstack_median_bootstrap](#), [90](#)
- [qxmatch](#), [89](#)
- [qxmatch2](#), [92](#)
- [radial_profile](#), [87](#)
- [random_coin](#), [80](#)
- [random_pdf](#), [80](#)
- [randomi](#), [80](#)
- [randomm](#), [80](#)
- [randpos_power](#), [89](#)
- [randpos_power_circle](#), [89](#)
- [randpos_uniform](#), [89](#)
- [randsrc](#), [92](#)
- [range](#), [34](#)
- [read_args](#), [47](#)
- [read_filter_db](#), [88](#)
- [rebin](#), [81](#)
- [reform](#), [44](#)
- [regex_extract](#), [60](#)
- [regex_match](#), [58](#)
- [regex_match_any_of](#), [58](#)
- [regex_replace](#), [61](#)
- [remcol](#), [92](#)
- [remove](#), [45](#)
- [remove_extension](#), [69](#)
- [replace](#), [55](#)
- [replace_block](#), [65](#)
- [replace_blocks](#), [65](#)
- [replicate](#), [44](#)
- [reverse](#), [38](#)
- [rgen](#), [79](#)
- [rgen_log](#), [79](#)
- [rotate](#), [87](#)
- [rotation_matrix](#), [82](#)
- [round](#), [78](#)
- [run_dim](#), [80](#)
- [run_dim_idx](#), [80](#)
- [scale](#), [87](#)
- [scale_matrix](#), [82](#)
- [seconds_str](#), [77](#)
- [sed2flux](#), [88](#)
- [sed_convert](#), [88](#)
- [sex2deg](#), [89](#), [92](#)

shift, 38
shrink, 86
shuffle, 80
sigma_clip, 81
sign, 79
sin, 78
sinh, 78
solve_symmetric, 83
sort, 37
spawn, 84
split, 56
sqr, 79
sqrt, 78
start_with, 62
strn, 51
strn_sci, 51
strna, 51
strna_sci, 51
subregion, 86
subsrc, 92
system_var, 66

tan, 78
tanh, 78
template_fit, 90

template_observed, 90
tgamma, 78
thread::pool, 84
thread::sleep_for, 84
time_str, 77
today, 77
tolower, 55
total, 80
toupper, 55
translate, 86
translation_matrix, 82
transpose, 82
trim, 54

uindgen, 36
uJy2lsun, 88
uJy2mag, 88
uniq, 41
upper_bound, 39

vuniverse, 88

warning, 77
where, 38
wrap, 57